М.Ю. Смоленцев

# Программирование на языке Ассемблера для 32/64-разрядных микропроцессоров семейства 80x86

Учебное пособие

часть 1

Иркутск 2009

УДК 004.43
ББК 32.973-018.7
С 50

**Смоленцев М.Ю.**

**С 50** Программирование на языке Ассемблера для 32/64-разрядных микропроцессоров семейства 80x86: Учебное пособие в 3-х частях. Часть 1. – Иркутск: ИрГУПС, 2009. – 192 с.

Рекомендовано Сибирским региональным учебно-методическим центром высшего профессионального образования для межвузовского использования в качестве учебного пособия для студентов специальностей 210700 – «Автоматика, телемеханика и связь на ж.-д. транспорте», 101800 – «Электроснабжение железных дорог» дневной и заочных форм обучения при изучении курсов «Микропроцессорные информационно-управляющие системы», «Программно-математическое обеспечение микропроцессорных систем» для специальности 071900 – «Информационные системы и технологии» при изучении курса «Периферийные устройства вычислительной техники», а также для курсового и дипломного проектирования.

# ВСТУПЛЕНИЕ

Учебное пособие по программированию на языке ассемблер для компьютеров, построенных на базе 32/64-разрядных микропроцессоров семейства 80x86, и методам программирования.

*Ассемблер* (Assembler) – язык программирования, понятия которого отражают архитектуру электронно-вычислительной машины. Язык ассемблера – символьная форма записи машинного кода, использование которого упрощает написание машинных программ. Для одной и той же ЭВМ могут быть разработаны разные языки ассемблера. В отличие от языков высокого уровня, в котором многие проблемы реализации алгоритмов скрыты от разработчиков, язык ассемблера тесно связан с системой команд микропроцессора. Для идеального микропроцессора, у которого система команд точно соответствует языку программирования, ассемблер вырабатывает по одному машинному коду на каждый оператор языка. На практике для реальных микропроцессоров может потребоваться несколько машинных команд для реализации одного оператора языка.

Язык ассемблера обеспечивает доступ к регистрам, указание методов адресации и описание операций в терминах команд процессора. Язык ассемблера может содержать средства более высокого уровня: встроенные и определяемые макрокоманды, соответствующие нескольким машинным командам, автоматический выбор команды в зависимости от типов операндов, средства описания структур данных. Главное достоинство языка ассемблера – «приближенность» к процессору, который является основой используемого программистом компьютера, а главным неудобством – слишком мелкое деление типовых операций, которое большинством пользователей воспринимается с трудом. Однако язык ассемблера в значительно большей степени отражает само функционирование компьютера, чем все остальные языки.

*Оптимальной* можно считать программу, которая работает правильно, по возможности быстро и занимает, возможно, малый объем памяти. Кроме того, ее легко читать и понимать; ее легко изменить; ее создание требует мало времени и незначительных расходов. В идеале язык ассемблера должен обладать совокупностью характеристик, которые бы позволяли получать программы, удовлетворяющие как можно большему числу перечисленных качеств.

Язык ассемблера обладает двумя принципиальными преимуществами: во-первых, написанные на нем программы требуют значительно меньшего объема памяти, во-вторых, выполняются гораздо быстрее, чем программы-аналоги, написанные на языках программирования высокого уровня. Кроме того, знание языка ассемблера облегчает понимание архитектуры компьютера и работы его аппаратной части, то, чего не может дать знание *языков высокого уровня* (ЯВУ). В настоящее время большинство программистов разрабатывает программы в средах быстрого проектирования (Rapid Application Development) когда все необходимые элементы оформления и управления создаются с помощью готовых визуальных компонентов. Это существенно упрощает процесс программирования. Однако, нередко приходится сталкиваться с такими ситуациями, когда наиболее мощное и эффективное функционирование отдельных программных модулей возможно только в случае написания их на языке ассемблера (ассемблерные вставки). В частности, в любой программе, связанной с выполнением многократно повторяющихся циклических процедур, будь это циклы математических вычислений или вывод графических изображений, целесообразно наиболее времяемкие операции сгруппировать в программируемые на языке ассемблера субмодули. Это допускают все пакеты современных языков программирования высокого уровня, а результатом всегда является существенное повышение быстродействия программ.

Языки программирования высокого уровня разрабатывались с целью возможно большего приближения способа записи программ к привычным для пользователей компьютеров тех или иных форм записи, в частности математических выражений, а также чтобы не учитывать в программах специфические технические особенности отдельных компьютеров. Язык ассемблера разрабатывается с учетом специфики процессора, поэтому для грамотного написания программы на языке ассемблера требуется, в общем, знать архитектуру процессора используемого компьютера. Однако, имея в виду преимущественное распространение IBM-совместимых персональных компьютеров и готовые пакеты программного обеспечения для них, об этом можно не задумываться, поскольку подобные заботы берут на себя фирмы-разработчики специализированного и универсального программного обеспечения.

Ответы на контрольные вопросы *необязательно* находятся в той главе, под которой они помещены. Иногда придется просмотреть и все пособие.

Автор придерживается мнения, что программирование на языке ассемблера – это искусство, которое основано на нескольких основных законах и включает в себя большое количество практических приемов и правил. Математических выкладок вы встретите очень мало, зато приводятся разнообразные примеры программ и всячески пропагандируется быстрая прикидочная оценка параметров и характеристик, которую желательно производить «в уме».

Вы не будете читать введение, не понимая языка, на котором оно написано. Это замечание относится к любой книге (исключая книжки-картинки). Чтобы получить от пособия максимум полезного, у Вас должны быть определенные знания, и Вы уже должны уметь делать нечто большее, чем просто читать. Данное пособие отнюдь не предназначено служить введением в программирование для ЭВМ. Предполагается, что читатель имеет некоторый опыт в этой области. Читатель должен:

1. иметь представление о том, как работает компьютер с заложенной в него программой: не столько о том, как работает электроника, сколько о том, как команды могут сохраняться в памяти машины и затем последовательно выполняться. Было бы полезно предварительное знакомство с любым из языков программирования;
2. знать хоть отчасти общепринятый компьютерный жаргон, например такие слова, как «память», «регистры», «биты», «плавающая точка», «переполнение», «кодировка ASCII», «файлы», «листинг»;
3. самостоятельно написать и отладить хотя бы две программы.

## Введение в язык ассемблера

*Язык программирования* – это система обозначений для описания данных и алгоритмов их обработки на компьютере. Программы для первых вычислительных машин составлялись на простейшем из языков программирования – машинном коде, при помощи только двух символов: нуля и единицы. Первое время программы писали в двоичном коде, но вскоре, программисты придумали себе облегчение – программы стали писать не в двоичной, а в шестнадцатеричной системе счисления. Для перевода из двоичной в шестнадцатеричную систему счисления каждые четыре двоичные цифры заменяют одной шестнадцатеричной цифрой (глава «Представление данных»). Например, $(1010\ 0010\ 0000\ 0111)_2 = (A207)_{16}$.

Программа на машинном коде имеет вид таблицы из цифр, каждая ее строчка соответствует одному оператору – машинной команде, которая является приказом компьютеру выполнить определенные действия. Конструкциями машинного кода являются константы и команды. Команда разделяется на группы бит (или поля). Первые несколько бит это поле – *код операции* (также **оп**ерационный **код**, *опкод* – англ *operation code*), которое показывает, что должен делать компьютер (складывать, умножать и тому подобное). Остальные поля, называемые *операндами*, идентифицируют требуемую команде информацию, показывают, где именно в памяти компьютера находятся нужные числа (слагаемые, сомножители и тому подобное) и куда следует поместить результат операции (сумму, произведение и так далее). Операнд может содержать данные, часть адреса, косвенный указатель на данные или другую информацию, относящуюся к обрабатываемым командой данным.

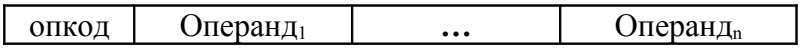



***Рис. 1. Общий формат команды***

Например, команда перемещения для микропроцессора 80x86 выглядит так: C605EF00400005. C605 – опкод операции перемещения. По такой команде компьютер помещает число 05 (две последние цифры 05) в ячейку памяти с номером 004000EF (цифры EF004000).

## Концепция Джона фон Неймана

Джон фон Нейман (John von Neumann) – профессор математики в Принстонском Институте сложных исследований – помогал проектировать компьютер EDVAC (Electronic Discrete Variable Automatic Computer – Электронный автоматический компьютер с дискретными переменными). В статье «Предварительное обсуждение логической конструкции электронной вычислительной машины», опубликованной в 1946 в соавторстве с Артуром Берксом (Arthur Burks) и Германом Гольдстайном (Herman Gold-

stine), он описал некоторые особенности компьютеров. Впоследствии оказалось, что концепция Джона фон Неймана настолько мощна и универсальна, что используется и по сей день в современных компьютерах:

1. компьютер должен состоять из следующих модулей: управляющий блок, арифметико-логический блок, оперативная память, блоки ввода/вывода;
2. строение компьютера не должно зависеть от решаемой задачи, программа должна храниться в памяти компьютера;
3. команды и данные должны храниться в одной и той же памяти;
4. память делится на ячейки одинакового размера, порядковый номер ячейки считается ее адресом;
5. программа состоит из серии элементарных инструкций, которые обычно не содержат прямого значения операнда (указывается только его адрес), поэтому программа не зависит от обрабатываемых данных. Инструкции выполняются одна за другой в том порядке, в котором они находятся в памяти;
6. для изменения порядка выполнения инструкций используются инструкции условного и безусловного перехода;
7. инструкции и данные (то есть операнды, результаты или адреса) представляются в виде двоичных сигналов и в двоичной системе счисления.

Двоичная система счисления используется в цифровых устройствах по следующим причинам:

- наименьшее количество возможных состояний элемента ведет к упрощению конструкции в целом, а также повышает помехоустойчивость и скорость работы;
- двоичная система счисления позволяет упростить операции сложения и умножения – основных действий над числами.

Составление программ на машинном коде – достаточно тяжелая и кропотливая работа, требующая чрезвычайного внимания и высокой квалификации программиста. Программист должен был удержать в своей памяти, где у него находятся переменные, где код программы, взаимосвязь между отдельными частями программ, но и объем памяти машины был не очень большой. Программист должен был помнить двоичные комбинации для каждого кода операции, адреса и константы, ввести их в память компьютера в правильном порядке. В программе на машинном коде очень легко ошибиться и очень трудно отыскать ошибку. Трудно расширять или сокращать уже написанные программы. Чтобы облегчить и повысить производительность труда, были созданы специальные программы *ассемблеры* (англ. *assemble* – собирать), которые выполняли рутинную работу по переводу символических команд СЛОЖИТЬ, ВЫЧЕСТЬ, ПЕРЕМЕСТИТЬ в нули и единицы машинного кода, а также разработан язык программирования – *ассемблер*. Язык ассемблера позволяет

составлять программы, используя для обозначения команд и символьных имен легко запоминающуюся мнемонику. Символьные имена выбираются программистом и служат для обозначения ячеек памяти и переменных.

Разные типы процессоров имеют разные наборы команд. Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется *языком программирования низкого уровня*. В данном случае «низкий уровень» не значит «плохой». Имеется в виду, что операторы языка близки к машинному коду и ориентированы на конкретные команды процессора.

Языком низкого уровня является язык ассемблера, который представляет каждую команду машинного кода, но не в виде двоичных чисел, а с помощью условных символьных обозначений, называемых *мнемониками*.

Для написания программ для человека был бы очень удобен его родной естественный язык (русский, английский или японский), но, к сожалению, для микропроцессора он *пока* непонятен. Единственный язык, который понимает микропроцессор – машинный код. Поскольку микропроцессоры имеют дело с цифровыми сигналами, команды машинного кода представляют собой двоичные коды. Микропроцессор конструируется таким образом, чтобы обеспечивалось распознавание конкретной группы кодов, которая называется *системой команд*.

Человеку нелегко пользоваться машинным кодом, поскольку смысл кода команды не ясен без соответствующего справочника. Можно заменить код каждой машинной команды коротким именем, называемым мнемоническим кодом. Например, код 01000000b или 40h для микропроцессора x86 означает «увеличить содержимое регистра EAX на единицу», выглядит как «INC EAX». Мнемонической командой служит трёх- или четырёхбуквенное сокращение английского глагола (см. табл. 1).

*Таблица 1*

**Мнемоники команд**

| Мнемоника | Смысл команды | Производное английское слова |
|---|---|---|
| jmp | продолжать выполнение с нового адреса в памяти | ***jump*** – прыгнуть |
| mov | переместить данные | ***move*** – переместить |
| sub | получить разность двух значений | ***substract*** – вычесть |
| xchg | обменять значения в регистрах/ ячейках памяти | ***exchange*** – обменять |

От ассемблера к ассемблеру меняется синтаксис аргументов, но мнемоники, обычно, остаются одинаковыми. В данном случае нет нужды помнить машинные коды каждой команды, а смысл каждой команды запоминается легче, чем ее код.

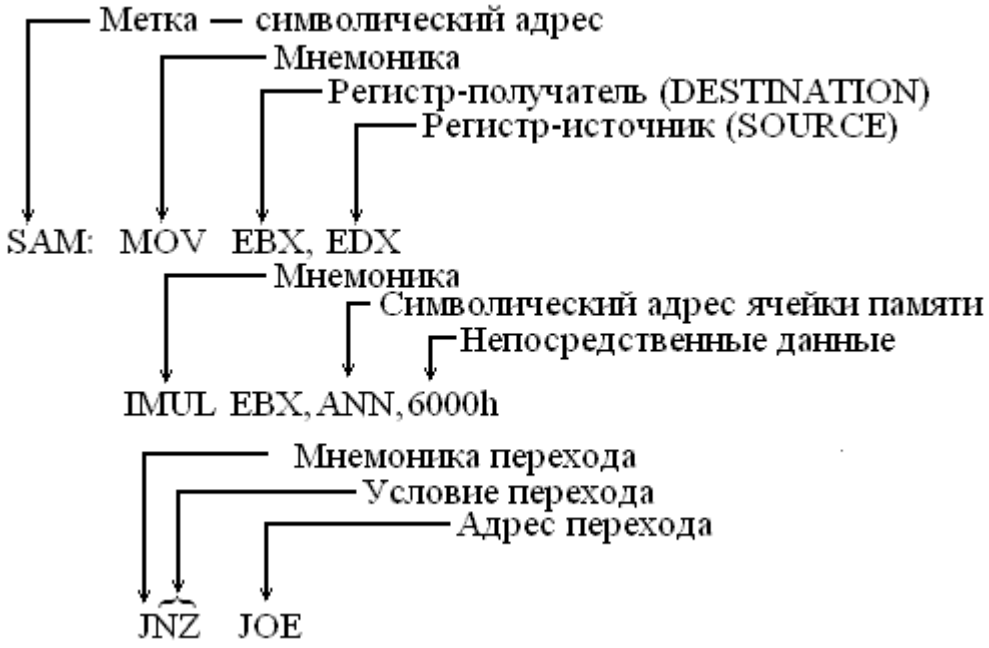



***Рис. 2. Примеры ассемблерных команд МП IA32/64***

Машинный код определяется главным образом конструкцией кристалла микропроцессора и не подлежит изменению. Мнемоника языка ассемблера разрабатывается изготовителем микропроцессора с ориентацией на обеспечение удобства программирования и не зависит от конструкции микропроцессора.

Язык ассемблера – это не какой-то один конкретный язык программирования, а целый класс языков. Каждый микропроцессор имеет свой собственный машинный код и, следовательно, собственный язык ассемблера (разрабатываемый изготовителем микропроцессора). В данной книге будет рассмотрен язык ассемблера для микропроцессоров семейства IA32/64.

Язык ассемблер для микропроцессоров IA32/64 поддерживают два синтаксиса Intel и AT&T. Под Intel-синтаксис разработаны следующие ассемблеры: MASM (Macro Assembler – Microsoft Corporation), BASM, TASM (Build-in Assembler, Turbo Assembler – Borland Inc), ASM-86 (Intel Corporation), FASM (Flat Assembler – Tomasz Grysztar), LZASM (lazzy assebmler – Степан Половников), WASM (Open Watcom Assembler – фирма Watcom), HLASM, HLA (High Level Assembler – IBM), NASM (NetWide Assembler – Simon Tatham, Julian Hall, Peter Anvin), YASM (Yet Another Assembler – Peter Johnson, Michael Urman), RosAsm (ReactOS Assembler), GoAsm (Jeremy Gordon) и т.д. Синтаксис AT&T используют AS (UNIX assembler) и GAS (GNU assembler).

Все вышеперечисленные ассемблеры включают стандартные мнемонические команды, в том числе команды арифметических операций для чисел с плавающей запятой для сопроцессора. Несмотря на имеющиеся различия между этими ассемблерами, их структуры и форматы команд языков в значительной мере совместимы.

В данном пособии будет рассматриваться программирование на языке Macro Assembler под 32-разрядную операционную систему Windows.

Любой язык программирования задается четырьмя компонентами: *алфавитом*, *синтаксисом*, *словарем* и *семантикой*.

Подобно естественным языкам, язык ассемблер строится над алфавитом основных символов (в котором записывается программа) в виде иерархической системы грамматических элементов с заданными на них отношениями (подобно словам, словосочетаниям и предложениям в естественном языке, связанным синтаксическими правилами). Элементы низшего уровня, образованные цепочками основных символов, называются *лексемами*.

*Алфавит* – это набор различных символов: букв, цифр, специальных знаков и т.п. Алфавит языка ассемблера состоит из букв *латинского* алфавита, цифр, парных ограничителей (скобок), разделителей (знаков препинания) и некоторых знаков операций. В связи с ограниченностью алфавита существуют правила кодирования основных символов комбинациями знаков, воспринимаемых входными устройствами машины. Основные классы лексем: *нумералы* для изображения чисел, *литералы* для изображения текстов, *идентификаторы* для обозначения различных объектов программы. Основные объекты языка ассемблер: *переменные*, *метки* (наименования различных частей программы) и *процедуры* (функциональные обозначения). Смысл и назначение некоторых идентификаторов фиксируется описанием языка (закрепленные слова или *словарь* языка ассемблер).

Алфавит языка ассемблера содержит такие символы:

- все строчные и прописные буквы латинского алфавита;
- десять арабских цифр: 0, 1, 2, …, 9;
- специальные знаки: « » (пробел), «?», «!», «.» (точка), «,» (запятая), «;» (точка с запятой), «:» (двоеточие), «@», «_» (знак подчеркивания), «’», «”», скобки полукруглые и квадратные, «/», «\», «>», «<», «*».

*Синтаксис* – это совокупность правил образования конструкций языка из символов, определенных алфавитом. Например, правило образования одной из конструкций языка ассемблера – *идентификатора*, или *имени*, заключается в следующем: идентификатор – это последовательность от одной до двухсот латинских букв, цифр и знаков «?», «.», «@», «_», «$», обязательно начинающаяся не с цифры; точка может быть только первым символом в имени; в зависимости от настроек, строчные и прописные буквы в имени могут не различаться.

Примеры идентификаторов: **A, A12345, ALFA, _67890, INDEX**.

*Словарь* – совокупность заранее оговоренных служебных слов: названия команд, названия директив ассемблера, названия регистров микропроцессора; в словарных словах строчные и прописные буквы не различаются.

*Семантика* – множество правил, позволяющих уточнить синтаксически правильные конструкции языка и определить их смысловое значение. Иными словами, зная и синтаксис, и семантику языка, можно понять, какие последовательности символов алфавита (тексты) имеют смысл, а какие – бессмысленны с точки зрения данного языка.

Программы, написанные на языке ассемблера, представляют собой последовательность приказов (инструкций, операторов). Эти операторы предварительно обрабатываются при помощи *транслятора* (программы, которая заменит каждый оператор языка ассемблера на соответствующую ему группу машинных кодов).

Однозначное преобразование одной машинной инструкции в одну команду языка ассемблера называется *транслитерацией*. Так как наборы инструкций для каждой модели процессора отличаются, каждой конкретной компьютерной архитектуре (платформы RISK, PowerPC, Alpha, Intel, Macintosh) соответствует свой язык ассемблера, и написанная на нем программа может быть использована только в этой среде.

Любую программу можно написать на языке ассемблера. С помощью ассемблера создаются очень эффективные и компактные программы, так как разработчик получает доступ ко всем возможностям процессора. Однако время и усилия, затраченные на ее написание, отладку и сопровождение будут гораздо больше того времени, которое ушло бы на реализацию такой программы на языке высокого уровня.

Кроме того, написание программы на языке ассемблера существенно ограничивает возможность переноса данной программы на другие вычислительные платформы. Гораздо выгоднее программировать на языке высокого уровня, для которого компилятор создает в одной программе фрагменты для наиболее распространенных платформ компьютеров. В таком случае можно основную часть программы реализовать на языке высокого уровня, а наиболее важные или критичные по быстродействию части реализовывать в виде вставок на языке ассемблера.

По мнению разных авторов, на языке ассемблера целесообразно разрабатывать такие части программы или программного комплекса, как, например, процедуры ввода/вывода низкого уровня, процедуры обработки прерывания и т.п.

На языке ассемблера также разрабатываются небольшие системные приложения, драйвера устройств, модули стыковки с нестандартным оборудованием, когда важнейшими требованиями становится компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам. Использование языков высокого уровня, имеющих достаточно большие ограничения, в этих случаях хотя и возможно, но весьма трудоемко. В некоторых областях, например в машинной графике, на языке ассемблера пишутся библиотеки, эффективно реализующие требующие интенсивных вычислений алгоритмы обработки изображений.

Первым ассемблером и одновременно первым интерпретатором стал псевдокод и набор инструкций Short Code, разработанный в 1949 г. американцами Пресом Экертом и Джоном Мошли для ЭВМ BINAC. Решение любой задачи вначале записывалось математическими уравнениями. Те, в свою очередь, посимвольно транслировались в коды: из «a=b+c» в «S0 03 S1 07 S2». На заключительном этапе коды приобретали двоичный вид, а каждая строка после ввода автоматически выполнялась. Первая практическая задача, которую решил ассемблер, – расчет таблиц артиллерийской стрельбы для американских баллистиков. Ассемблеры на мнемонических кодах («MOV», «ADD») появились только в середине 50-х. Авторы языка ассемблер более известны изобретением вычислительных машин на вакуумных трубках: ENIC (1946 г.), BINAC (1949 г.) и прямого предка современных компьютеров UNIVAC I (1951 г.).

## Контрольные вопросы

1. Почему язык ассемблера называют языком программирования низкого уровня?
2. В каком виде можно представить машинные команды?
3. Что означают понятия семантика и синтаксис в языках программирования?
4. Что такое транслитерация?
5. Какова функция программы-транслятора?
6. Что такое команды и что такое данные?
7. Назовите основные компоненты компьютера, без которых невозможна его работа.

# ГЛАВА 1
# МИКРОПРОЦЕССОРЫ ФИРМЫ INTEL

*Intel Technologies Incorporated* – американская корпорация, крупнейший в мире производитель микропроцессоров, оборудования для персональных компьютеров, компьютерных систем и средств связи.

Основана в 1968 г. Робертом Нойсом и Гордоном Муром. Тогда же к ним присоединился Эндрю Гроув. Цель нового предприятия – разработка на базе полупроводниковых технологий более дешевой альтернативы запоминающим устройствам на магнитных носителях.

Важнейшим компонентом любого персонального компьютера, его «мозгом» является *микропроцессор*, который управляет работой компьютера и выполняет большую часть обработки информации. Микропроцессор представляет собой сверхбольшую интегральную схему, степень интеграции которой определяется размером кристалла и количеством реализованных в нем транзисторов. Базовыми элементами микропроцессора являются транзисторные переключатели, на основе которых строятся, например, регистры, представляющие собой совокупность устройств, имеющих два устойчивых состояния и предназначенных для хранения информации и быстрого доступа к ней. Количество и разрядность регистров во многом определяют архитектуру микропроцессора.

Выполняемые микропроцессором команды предусматривают, как правило, арифметические действия, логические операции, передачу управления (условную и безусловную) и перемещение данных (между регистрами, оперативной памятью и портами ввода/вывода). С внешними устройствами микропроцессор может общаться благодаря своим шинам адреса, данных и управления, выведенным на специальные контакты корпуса микросхемы. Разрядность внутренних регистров микропроцессора может не совпадать с количеством внешних выводов для линий данных, например, микропроцессор с 32-разрядными регистрами может иметь только 16 внешних линий данных. Объем физически адресуемой микропроцессором памяти однозначно определяется разрядностью внешней шины адреса как $2^N$, где N – количество адресных линий.

## 1.1. Основные тенденции в развитии микропроцессоров

*Закон Мура*. В 1965 г. журнал Electronics (vol. 39, № 8) в рубрике «Эксперты смотрят в будущее» выпустил статью «Cramming more components onto integrated circuits», в которой будущий основатель Intel Гордон Мур предсказал, что *количество структурных элементов (транзисторов) на кристалле процессора, а следовательно и вычислительная мощность процессоров будут удваиваться каждые полтора-два года*. Прошло сорок лет, а закон Мура продолжает действовать (таблица 1.1).

*Понижение рабочего напряжения.* По мере развития процессорной техники происходит постепенное *понижение рабочего напряжения*. Ранние

модели Intel x86 имели рабочее напряжение 12 и 5 В, с переходом к процессорам Pentium оно было понижено до 3,3 В, в настоящее время оно менее 1 В. Понижение рабочего напряжения позволяет уменьшить расстояние между структурными элементами в кристалле процессора до долей *микрона* (далее по тексту – μ), не опасаясь электрического пробоя. Уменьшение размера, в свою очередь, позволяет разместить еще большее количество элементов на кристалле. Пропорционально квадрату напряжения уменьшается и тепловыделение в процессоре, а это позволяет увеличивать скорость его работы без угрозы пробоя, что в свою очередь приводит к *увеличению рабочей частоты ядра процессора*. Первые микропроцессоры x86 работали с частотой не выше 1МГц, рабочая частота Pentium 4 доходит до 5ГГц.

*Таблица 1.1*

**Закон Мура**

| Микропроцессор | Год выпуска | Число транзисторов |
|---|---|---|
| i4004 | 1971 | 2 300 |
| i8008 | 1972 | 2 500 |
| i8080 | 1974 | 5 000 |
| i8086 | 1978 | 29 000 |
| i80286 | 1982 | 120 000 |
| i80386 | 1985 | 275 000 |
| i80486 | 1989 | 1 180 000 |
| Pentium | 1993 | 3 100 000 |
| Pentium II | 1997 | 7 500 000 |
| Pentium III | 1999 | 24 000 000 |
| Pentium 4 | 2000 | 42 000 000 |
| Itanium | 2002 | 220 000 000 |
| Itanium 2 | 2003 | 410 000 000 |
| Itanium (Montecito) | 2005 | 1 720 000 000 |

*Увеличение разрядности процессора*. Разрядность процессора показывает сколько бит данных он может принять и обработать в своих регистрах за один такт. Микропроцессор i4004 был 4-разрядным, i8008 – 8-разрядный, i8086 уже 16-разрядный, начиная с i80386 микропроцессоры становятся 32-разрядными, Pentium'ы хотя и остаются 32-разрядными, но работают уже с 64-разрядной шиной данных. Компьютеры становятся двухъядерными, то есть два 32-разрядных процессора параллельно обрабатывают 64 разряда данных. В Pentium Pro появляется 128-разрядная внешняя шина данных, и компьютеры становятся уже четырехъядерными. Ожидается появление полностью 64-разрядных микропроцессоров.

## 1.2. Начало

Микропроцессор, как универсальный блок обработки информации, был разработан в 1962 г. Время поступления на рынок первых микро-ЭВМ совпало с завершением разработки однокристальных процессоров фирмой

Intel. В 1969 г. Intel получил заказ на изготовление 12 типов микросхем для калькуляторов различных моделей. Малый объем каждой партии увеличивал стоимость их разработки. Инженер Intel Тед Хоф сконструировал объединенную микросхему – универсальное логическое устройство, которое отыскивало и отбирало прикладные команды из полупроводниковой памяти. Являясь ядром набора из четырех микросхем, этот центральный вычислительный блок не только соответствовал требованиям заказа и мог использоваться не только во всех типах заказанных калькуляторов, но и найти самое разнообразное применение без каких-либо переделок. С 1971 года Intel начал промышленный выпуск микропроцессора i4004. Он представлял собой четырехразрядное параллельное вычислительное устройство. С его помощью можно было обрабатывать четырехразрядные двоичные числа, действия над более длинными операндами выполнялись по частям. i4004 выполнял 45 различных команд и, тем не менее, его возможности были сильно ограничены. Четыре бита позволяли кодировать только цифры и символы знаков арифметических операций, хотя этого и было достаточно для математических расчетов. i4004 применялся поначалу только в карманных калькуляторах. Позднее сфера его применения была расширена за счет использования в различных системах управления, например, i4004 встраивался внутрь светофоров. Одновременно с выпуском микропроцессоров i4004 было организовано производство специализированного набора микросхем: модулей управления вводом/выводом i4009, оперативных запоминающих устройств i4002, постоянных запоминающих устройств i4001 и других. Выбирая тот или иной комплект микросхем, разработчик мог построить небольшой калькулятор, управляемый программой, записанной в ПЗУ.

Intel, правильно предугадав перспективность микропроцессоров, продолжил интенсивные разработки, и один из его проектов в конечном итоге привел к крупному успеху, предопределившему будущий путь развития вычислительной техники. Им стал проект по разработке 8-разрядного микропроцессора i8008 (1972 г.). ЭВМ должны работать не только с цифрами, но также и с текстами. Использование 6 бит позволяет различать все цифры, а также большие и малые латинские буквы (можно закодировать $2^6$=64 символа). Но при этом остается мало значений для кодировки знаков пунктуации и управляющих символов, поэтому регистры i8008 сделали 8-разрядными ($2^8$ это уже 256 символов). По сравнению с i4004 микропроцессор i8008 имеет очень развитую систему команд. Первоначально i8008 предназначался только для управления терминалом. Но как только i8008 стал доступен разработчикам из других фирм, этот микропроцессор предоставил им безграничные возможности для творчества и новаторской деятельности. В продуктовых магазинах появились первые цифровые весы – микросхема преобразовывала вес продуктов в цены и считывала этикетки с покупаемых товаров. Новый микропроцессор внес революционные изменения во все сферы жизни – от медицинских инструментов

до кассовых систем ресторанов «быстрого питания», от бронирования авиабилетов до заправки топливом на бензоколонках.

Модификация этой микросхемы i8080 (1974 г.) – полноценное арифметико-логическое устройство, на базе которого было построено множество бытовых персональных компьютеров. Она же послужила прототипом для создания микропроцессора Z-80. i8008 был использован при создании любительской мини-ЭВМ (прообраз персонального компьютера) «Altair-8800», которая была создана в конце 1974 г. фирмой *MITS* (*Micro Instrumentation and Telemetry Systems*). Для этого компьютера *Бил Гейтс* написал один из своих первых интерпретаторов языка Basic-80.

i8080 требовалось три напряжения питания и два поступающих извне тактовых сигнала с уровнем 12В и частотой до 2 МГц с точно выдержанной задержкой между ними. i8085 имел систему команд i8080, а также ряд аппаратурных усовершенствований (единственное напряжение питания +5В, отсутствие «проблемы тактов» и так далее), что упрощало применение микропроцессоров.

i8080 и i8085 относятся к классу 8-битных микропроцессоров, каждый следующий микропроцессор Intel становился все более сложным и гибким.

## 1.3. Микропроцессор i8086

В 1976 г. Intel закончил разработку 16-разрядного микропроцессора i8086. Он имел достаточно большую разрядность регистров (16 бит), 16-битную шину данных и 20-битную системную шину адреса, за счет чего мог адресовать до 1 Мбайта ($2^{20}$=1.048.576≈$10^6$) оперативной памяти. Тактовая частота 4 МГц. Машинные коды i8086 несовместимы с кодами i8080, но уже содержат специальные команды для деления и умножения.

## 1.4. Микропроцессор i8088

Микропроцессор i8088 имеет 8-битную шину данных, но сохраняет все функциональные возможности микропроцессора i8086. Регистры микропроцессора i8088 полностью соответствуют регистрам i8086. Отличие этих микропроцессоров заключается только во внутренней реализации, а с точки зрения программирования они полностью идентичны. Переход к 8-битной шине данных позволил использовать широкораспространенные на тот период вспомогательные микросхемы и внешние устройства, разработанные для i8080.

В 1980 г., исходя из текущей рыночной ситуации, корпорация IBM решает построить собственный 16-разрядный компьютер, подобный популярному Apple, с похожим программным обеспечением. При конструировании компьютера был применен принцип открытой архитектуры: его составные части были универсальными, что позволяло модернизировать компьютер по частям. Для уменьшения затрат на создание персонального компьютера IBM использовала разработки других фирм в качестве составных частей для своего детища, в частности, микропроцессор Intel – i8088 и программное обеспечение Microsoft. Появление IBM PC (*персональный*

*компьютер*, **P**ersonal **C**omputer) в 1981 г. породило лавинообразный спрос на персональные компьютеры, которые стали теперь орудием труда людей самых разных профессий. Наряду с этим возник гигантский спрос на программное обеспечение и компьютерную периферию. На этой волне возникли сотни новых фирм, занявших свои ниши компьютерного рынка.

Также на основе микропроцессора i8088 в 1983 г. создан компьютер XT (компьютер с *расширенной технологией*, **Ex**tended **T**echnology).

Первоначально микропроцессор i8088 работал с частотой 4,77 МГц и имел быстродействие около 0,33 млн инструкций в секунду (*Million Instruction Per Second*, MIPS), однако впоследствии были разработаны его клоны, рассчитанные на более высокую тактовую частоту (например, 8 МГц).

## 1.5. Микропроцессор i80186

Микропроцессор i80186 (1982 г.) – однокристальное расширение микропроцессора i8086, с внутренним генератором синхронизации, логикой управления прерываниями, схемой таймеров, контроллерами *ПДП* (*прямого доступа к памяти*) и программируемыми схемами выбора кристалла. i80186 обладал более высоким быстродействием и дополнительными вычислительными возможностями, которые особенно важны при проектировании сложных микросхем. В основном i80186 нашел применение в микроконтроллерах.

## 1.6. Микропроцессор i80188

Модификация микропроцессора i80186 с 8-битной шиной данных.

## 1.7. Микропроцессор i80286

В 1982 г. был создан i80286, представлял собой улучшенный вариант i8086. Оснащенный встроенным устройством управления памятью, он стал первым микропроцессором, совместимым со своими предшественниками. Поддерживал несколько режимов работы с памятью: *реальный*, когда формирование адреса производится по правилам i8086, и *защищенный*, который аппаратно реализовывал многозадачность и управление виртуальной памятью. i80286 имел 24-битную системную шину адреса, поэтому мог адресовать до 16 Мбайт ($2^{24}$=16.777.216) оперативной памяти. На основе i80286 в 1984 г. создан компьютер AT (компьютер с *улучшенной технологией*, **A**dvanced **T**echnology).

## 1.8. Микропроцессор i80386

В 1985 г. Intel представила первый 32-разрядный микропроцессор i80386, аппаратно совместимый снизу вверх со всеми предыдущими микропроцессорами этой фирмы. Он был гораздо мощнее своих предшественников, имел 32-разрядную архитектуру (32-битные регистры, 32-бит-

ную шину данных и 32-битную системную адресную шину), мог прямо адресовать до 4 Гбайт ($2^{32}$=4.294.967.296) оперативной памяти. Первым компьютером, использующим этот микропроцессор, был Compaq DeskPro 386. 32-разрядная архитектура нового микропроцессора дополнена *расширенным устройством управления памятью* (*Memory Management Unit*, MMU). На тактовой частоте 16 МГц быстродействие i80386 составило примерно 6MIPS. Кроме того, был введен новый режим – *режим виртуального процессора i8086* (V86). В этом режиме могли одновременно выполняться несколько задач, предназначенных для i8086.

Первые i80386 пугали людей как своей работой, так и ценой. Intel решил выпустить продукт, более доступный пользователям с менее мощным (зато более дешевым) микропроцессором i80386SX (1988 г.). В очередной раз происходит ухудшение качества микропроцессора, как это уже случалось с i8088, с i80188, а затем произойдет с Celeron. i80386SX имел внутреннюю полностью 32-разрядную архитектуру, но использовал 16-разрядную внешнюю шину данных и 24-разрядную адресацию. Индекс микропроцессора SX происходит от **SIXTEEN** (*шестнадцать*) в соответствии с разрядностью шины данных. Чтобы не путать микропроцессор i80386 с менее мощной микросхемой i80386SX, Intel переименовал i80386 в i80386DX.

## 1.9. Сопроцессор

Первоначально математический сопроцессор, называемый также арифметическим сопроцессором, представлял собой специализированную интегральную схему, работавшую во взаимодействии с центральным микропроцессором. Данная микросхема была предназначена только для выполнения математических операций с плавающей точкой. Во всех микропроцессорах Intel от i80486DX и выше сопроцессор интегрирован на кристалл основного процессора.

Математический сопроцессор необходим при работе с векторной графикой (особенно трехмерной), электронными таблицами, пакетами САПР, специальными математическими пакетами и т.п. При работе же с базами данных или обычными текстовыми редакторами использование сопроцессора не дает ощутимых результатов. Он бесполезен и при работе с сетевыми операционными системами.

Первым математическим сопроцессором для персональных компьютеров IBM был i8087 компании Intel. Далее последовали i80287, i80387, i80487SX. Клоны сопроцессоров Intel выпускали такие фирмы, как ULSI, AMD, Cyrix (Texas Instruments), IIT, Chips. Начиная с процессоров пятого поколения, сопроцессор стал стандартным компонентом ядра процессора.

## 1.10. Кэш-память

*Кэш-память* (*cache memory*) – запоминающее устройство с малым временем доступа (в несколько раз меньшим, чем время доступа к основной оперативной памяти), используемое для временного хранения промежуточных результатов и содержимого часто используемых ячеек. Вообще кэшированием данных называется размещение данных в области памяти с более быстрым доступом. В качестве житейской аналогии можно привести библиотеку школьника, у которого нужные каждый день учебники лежат на рабочем столе, изредка читаемые классики стоят на книжной полке, а старые ненужные тетради сложены в ящиках. В случае необходимости время доступа к этим источникам будет разным, однако и вероятность того, что потребуется учебник или старая тетрадь, тоже разная.

В мире компьютерной памяти этот принцип применим потому, что более быстрая память обычно стоит существенно дороже более медленной, однако применение малого объема быстрой (но дорогой) кэш-памяти, в комплексе с большим объемом медленной (но дешевой) памяти, позволяет создать приемлемое по цене и скорости решение. Применение кэширования особенно эффективно, когда доступ к данным осуществляется преимущественно в последовательном порядке. Тогда после первого запроса на чтение данных, расположенных в медленной (кэшируемой) памяти можно заранее выполнить чтение следующих блоков данных в кэш-память для того, чтобы при следующем запросе на чтение данных почти мгновенно выдать их из кэш-памяти. Такой прием называется упреждающим чтением.

Упреждающее чтение применяется во всех современных жестких дисках, имеющих от 64 до 1024 Кбайт кэш-памяти, выполненной на основе динамической *RAM* (*Random Access Memory* – запоминающее устройство с произвольной выборкой). Считываемые с диска данные с некоторым запасом помещаются в кэш-память диска и определенное время там хранятся. При повторном обращении к тем же данным они считываются уже из кэш-памяти, что происходит в 10–1000 раз быстрее.

Кэширование данных применяется также в процессорах. Внутри кристалла процессора находится малый объем (от 1 до 1024 Кбайт) очень быстрой статической памяти, работающей на частоте процессора. Эта память используется для кэширования существенно более медленной оперативной памяти, выполненной на основе динамической RAM. Таким образом, в различных ситуациях одна и та же память может быть как кэшем, так и кэшируемой памятью.

Кэш-память также может быть организована в виде иерархической структуры. В случае процессоров семейства x86 характерно использование кэша первого уровня (Level 1 или L1-кэша), расположенного непосредственно на кристалле процессора, и более медленного кэша второго уровня (Level2 или L2-кэша), расположенного в другой микросхеме или вообще на другой плате. При этом кэш первого уровня кэширует L2-кэш, а тот, в свою

очередь, кэширует еще более медленную оперативную память. В RISC-процессорах зачастую используется L3-кэш и кэш более высоких порядков.

Существуют различные алгоритмы работы кэш-памяти, которые очень сильно влияют на эффективность процедуры кэширования. Помимо кэширования операций чтения данных можно выполнять кэширование записи данных (это называется *отложенной записью*, или *lazy write*, для жестких дисков и *обратной записью*, или *write back*, для процессоров). Применение отложенной записи еще больше увеличивает скорость работы диска, но повышает риск потери данных, которые не успели записаться из кэш-памяти в кэшируемую память, в случае внезапного краха системы.

## 1.11. Микропроцессор i80486

Микропроцессор i80486DX выпущен в 1989 г. В i80486 математический сопроцессор интегрирован в одном кристалле с основным процессором. С микропроцессором i80486 появилось понятие *конвейеризации вычислений* (глава «Архитектура микропроцессора i80x86»).

Микропроцессор i80486 стал дополняться внутренней кэш-памятью. *Кэширование* – это способ увеличения быстродействия системы за счет хранения часто используемых данных и кодов в так называемой «кэш-памяти первого уровня» (быстрой памяти), находящейся внутри микропроцессора. i80486 содержал блок встроенной кэш-памяти размером 8 Кбайт, который использовался для кэширования и кодов, и данных. Существовали модификации этого процессора: i80486SX, i80486DX, i80486DX2 и i80486SL. Тактовая частота до 120 МГц.

## 1.12. Семейство микропроцессоров Pentium

Микропроцессор Pentium выпускался с 1993 г. Имеет 32-разрядную архитектуру и 64-разрядную шину данных и адреса, за счет чего может адресовать до 16 Терабайт ($2^{64} \approx 16 \times 10^{18}$) оперативной памяти. К выходу этой книги в 2007 г. выпущена уже четвертая модификация микропроцессора Pentium – Pentium-4 с тактовой частотой 5 ГГц. *Скалярным* называют процессор с единственным конвейером, к этому типу относятся все процессоры Intel до 486 включительно. *Суперскалярный* (*superscalar*) процессор имеет более одного конвейера, что позволяет обрабатывать несколько инструкций параллельно. Pentium является двухпотоковым процессором (имеет два конвейера), процессоры от Pentium-II до Pentium-4 – трехпотоковые. В Pentium’е было введено раздельное кэширование кода и данных. Pentium содержит два блока кэш-памяти: один для кода и один для данных, каждый по 8 Кбайт. При этом становится возможным одновременный доступ к коду и данным, что увеличивает скорость работы компьютера.

### 1.12.1. Микропроцессор Pentium

Выпускался с 1993 г. Тактовая частота 66 и 60 МГц. По базовой регистровой архитектуре и системе команд он является 32-разрядным процессором, но имеет 64-битную шину данных. Шина адреса позволяет адресовать 4 Гигабайта физической памяти. Интерфейс рассчитан на применение внешнего вторичного кэша и внутреннего первичного с возможностью работы как со сквозной (WT), так и с обратной записью. Интерфейс позволяет объединять до двух процессоров на одной шине.

Pentium содержит два арифметико-логических устройства, благодаря которым две команды могут быть выполнены за один такт синхронизации. Pentium имеет также два раздельных кэша по 8 Кбайт: один – для команд, другой – для данных. Pentium – первый массовым процессор Intel с суперскалярной архитектурой и динамическим предсказанием переходов в исполняемых программах. В Pentium была существенно повышена производительность модуля вычислений с плавающей запятой, добавлена аппаратная поддержка самотестирования, текущего контроля производительности и расширенной отладки. Благодаря встроенному в Pentium контроллеру прерываний многопроцессорных систем получили распространение двухпроцессорные серверы и рабочие станции на его основе.

Pentium с тактовой частотой 66 МГц имел производительность около 112 MIPS. Выпускались версии Pentium второго поколения с тактовыми частотами от 75 до 200 МГц и напряжением питания 3,3 В, построенные с использованием 0,6- и 0,35-μ технологии.

### 1.12.2. Микропроцессор Pentium Pro

В 1995 г. Intel выпустил микропроцессор Pentium Pro. В его основе лежит комбинация технологий Dynamic Execution: многократное предсказание ветвлений (*multiple branch prediction*), анализ потоков данных (*data flow analysis*) и эмуляция выполнения инструкций (*speculative execution*). Глубина конвейера составляет 14 ступеней (в Pentium 5 ступеней), благодаря чему на исполнение одной команды в среднем приходится на 33 % меньше времени. В Pentium Pro впервые реализована архитектура *DIB* (*Dual Independent Bus*) – независимая двойная шина. Одна шина, работающая на частоте процессора, связывает процессор и встроенную кэш-память, другая, работающая на внешней тактовой частоте 60 или 66 МГц, – процессор и системную плату. В Pentium Pro впервые введена 128-разрядная внешняя шина данных.

В корпусе микросхемы размещены два кристалла – собственно процессор и кэш-память второго уровня объемом от 256 до 1024 Кбайт, работающая на частоте процессора. На кристалле процессора расположен 16-Кбайтный кэш. В семейство Pentium Pro входят микропроцессоры с тактовыми частотами от 150 до 200 МГц. Pentium Pro поддерживает 2-х и 4-процессорные конфигурации, а также позволяет строить системы с числом процессоров более 4 с применением технологий кластеризации. Pentium Pro имеет 36-разрядную шину адреса и способен адресовать до 65

Гбайт оперативной памяти. Недостатком объединения в одном корпусе двух отдельных микросхем (собственно процессора и кэш-памяти второго уровня) является высокая стоимость производства, поскольку при наличии дефекта только в одной из них приходится отбраковывать весь процессор.

Препятствием к дальнейшему повышению тактовой частоты являлась высокая стоимость изготовления микросхем кэш-памяти, способных работать на одной частоте с процессором. Микросхема Pentium Pro 150 выпускалась по 0,6-μ технологии, остальные процессоры с более высокой тактовой частотой выпускались с использованием 0,35-μ технологии. Напряжение питания процессора 3,3 В.

Процессор был оптимизирован для выполнения 32-разрядных инструкций и демонстрировал максимальную производительность под 32-разрядными операционными системами (Windows NT, OS/2, UNIX). 16-разрядные инструкции вызывали частую перезагрузку конвейера, что приводило к меньшей производительности этого процессора в 16- и 16/32-разрядных ОС (MS DOS, Windows 3.1/3.11, Windows 95).

Процессор Pentium Pro и наборы микросхем для него позволяли создавать многопроцессорные системы, основной областью применения этого процессора были высокопроизводительные сервера.

### 1.12.3. Микропроцессор Pentium MMX

С 1997 г. Intel начал производить процессор Pentium с технологией MMX – микропроцессор, в котором, впервые со времени выпуска i80386, произошло крупное расширение системы команд за счет включения 57 новых инструкций, разработанных для более эффективной работы с мультимедийными данными (MMX: *Multi Media Extensions*, расширения для мультимедиа). Новые инструкции ориентировались на параллельное исполнение повторяющихся последовательностей команд целочисленной арифметики, часто встречающихся при работе мультимедиа-приложений. В основе MMX лежал применяемый в процессорах цифровой обработки сигналов принцип SIMD (*Single Instruction – Multiple Data*, один поток команд – множество потоков данных), который состоял в применении одной инструкции для массива однородных данных. Для обработки MMX-инструкций используются регистры математического сопроцессора. Использование MMX-инструкций ускоряет работу программ обработки изображений, видео или звука на 50–400 %.

В Pentium MMX кэш первого уровня был расширен до 32 Кбайт и применялось более эффективное предсказание условных переходов, что позволило на 10–20 % повысить производительность процессора.

Pentium MMX выпускался с тактовой частотой от 120 до 266 МГц для настольных и мобильных систем. Процессоры использовали двойное напряжение питания: 2,8 В для ядра процессора (от 2,45 до 1,9 В в мобильных системах) и 3,3 В для схем обрамления, и были построены с использованием 0,35- и 0,25-μ технологий.

### 1.12.4. Микропроцессор Pentium II

Pentium II был представлен в 1997 г. Добавление в ядро P6 блока обработки MMX-инструкций и вынесение из корпуса процессора кэш-памяти второго уровня резко снизило стоимость производства.

В Pentium II, объем кэш-памяти первого уровня увеличен до 32 Кбайт. 512 Кбайт кэш-памяти второго уровня выполнены на отдельных серийных микросхемах BSRAM и смонтированы вместе с процессором на небольшой печатной плате. Использовалась архитектура независимой двойной шины DIB, но частота ее работы между процессором и кэш-памятью второго уровня уменьшена до половины от внутренней тактовой частоты процессора. Шаг назад по сравнению с Pentium Pro – снижение объема адресуемой памяти до 4 Гбайт (32-разрядная адресная шина), а также возможность создания только двухпроцессорных систем. Упрощение системной архитектуры позволило снизить цену на процессоры и добиться массового распространения Pentium II. Pentium II первого поколения выпускался с использованием 0,35-μ технологии, имели тактовые частоты от 233 до 333 МГц и содержали 7,8 млн транзисторов (без учета кэш-памяти второго уровня). Напряжение питания первого поколения процессоров составляло 2,8 В.

Intel учел недостатки Pentium Pro, поэтому процессор Pentium II был оптимизирован как для 32-, так и для 16-разрядных приложений за счет сегментирования кэш-памяти первого уровня.

Дальнейшее развитие Pentium II привело к появлению трех различных семейств процессоров на основе архитектуры P6: Deschutes, Celeron (включая Mendocino) и Pentium II Xeon.

С 1998 г. Intel выпускает Pentium II второго поколения. Существовали модели с тактовыми частотами от 350 до 450 МГц для настольных и мобильных систем. За счет использования 0,25-μ технологии и снижения напряжения питания до 2 В уменьшены размеры кристалла, его энергопотребление и себестоимость.

### 1.12.5. Микропроцессор Celeron

Выпускался с 1998 г. Celeron представлял собой Pentium II второго поколения, лишенный кэш-памяти второго уровня. Изначально процессор позиционировался как экономичное решение. Celeron выпускался по 0,25-μ технологии с тактовыми частотами 266 и 300 МГц.

Системная шина между памятью и процессором является узким местом при работе с большим коэффициентом умножения частоты и отсутствие кэш-памяти второго уровня существенно снижает производительность системы. С другой стороны, производительность компьютера сильно зависит от типа решаемой задачи. Приложения, активно работающие с большими объемами памяти (базы данных, инженерные и научные программы), на процессоре Celeron исполнялись неэффективно, в то время как офисные

приложения и большинство компьютерных игр демонстрировали на нем производительность, немного уступавшую оригинальному Pentium II.

Чтобы ускорить продвижение Celeron на рынок и преодолеть его недостаточную производительность, Intel форсировал выпуск процессора со встроенной памятью 128 Кбайт с тактовыми частотами 300 и 333 МГц, который был назван Celeron 300A и 333, соответственно. Процессор выпускался по 0,25-μ технологии. Так как кэш-память второго уровня была расположена на одном кристалле с процессором, ее частота повышена до полной частоты процессора. В результате наблюдался значительный прирост производительности новых Celeron.

С 1999 г. выпускается процессор Celeron (Celeron II) с ядром Coppermine (0,18μ, питание 1,5 В). Размер вторичного кэша 128 Кбайт, частота шины 66 МГц, поддержка инструкций SSE.

### 1.12.6. Микропроцессор Pentium III

Процессор Pentium III (1999 г.) – дальнейшее развитие Pentium II. Главное отличие этого микропроцессора – расширение набора инструкций командами SSE (*Streaming SIMID Extension*), основанных на блоке 128-разрядных регистров XXM. Этот блок позволяет одной инструкцией выполнять операции сразу над четырьмя комплектами 32-разрядных операндов в формате с плавающей точкой (одинарная точность). При выполнении новых инструкций оборудование традиционного FPU/MMX не используется, что позволяет смешивать инструкции MMX с инструкциями над операндами с плавающей точкой. Появились новые возможности управления кэшированием. По возможностям мультипроцессорных конфигураций эти процессоры были аналогичны своим предшественникам Pentium II и Pentium II Xeon. Частота ядра начинается с 500 МГц, частота системной шины 100 и 133 МГц. Вторичный кэш в первых моделях 512 Кбайт в виде отдельных микросхем.

### 1.12.7. Микропроцессор Pentium 4

Процессор Pentium 4, по микроархитектуре принадлежит к 7 поколению процессоров Intel. Содержит расширение команд SSE2. По набору регистров повторяет процессор Pentium III. Микроархитектура процессора – NetBurst, разработана с учетом высоких частот как ядра (1,4 ГГц) так и системной шины (400 МГц). На одном кристалле расположена кэш-память двух уровней. Шина адреса 36-разрядная, что позволяет адресовать 64 Гбайта памяти, из которых кэшируется 4 Гбайта. Шина данных 64-разрядная. Напряжение питания 1,6 В.

АЛУ процессора работает на удвоенной частоте ядра. За каждый такт процессора выполняются две основные целочисленные команды. Обеспечена более высокая пропускная способность потока команд через исполнительную часть процессора и уменьшены различные задержки.

Гиперконвейер состоит из 20 ступеней. Цель увеличения конвейера – упрощение задач, реализуемых каждой из его ступеней и, как следствие, упрощение соответствующей аппаратной логики.

Улучшена технология динамического исполнения благодаря более глубокой «произвольности» в порядке исполнения кода и усовершенствованной системы предсказания переходов. Размер буфера меток и перехода увеличен до 4 Кбайт (Pentium III – 512 байт). Усовершенствован сам алгоритм предсказания. В результате вероятность правильного предсказания перехода возрастает до 95 %.

Отсутствует кэш команд первого уровня. Он заменен на кэш *трассы* (*trace cache*). *Трассами* называются последовательность микроопераций в которые были декодированы ранее выбранные команды. В этом кэше размещается до 12 К микроопераций. Кэш трасс доставляет исполнительному ядру до трех микроопераций за такт. Вторичный кэш, общий для инструкций и данных, имеет размер 256 Кбайт и 256-разрядную шину, работающую на частоте ядра. Первичный кэш данных (8 Кбайт) так же работает на частоте ядра.

Поддержка технологии Hyper Threading, которая на базе одного физического процессора позволяет моделировать несколько логических, каждый из которых имеет собственное архитектурное пространство.

## Контрольные вопросы

1. Сформулируйте закон Мура.
2. Какова роль процессора в компьютерной системе?
3. Почему операционные системы постоянно модифицируются?
4. Для чего нужна оперативная память?
5. Что означает понятие конвейеризации вычислений?
6. Какие микропроцессоры называются скалярными?
7. Что такое кэширование? Для чего нужна кэш-память?
8. Для чего в архитектуру Pentium был введен блок предсказания переходов?
9. Предположим, что имеется 20 линий адреса и 16 линий данных:
   а) определите адресное пространство памяти, если адресные пространства памяти и ввода-вывода различны;
   б) каков диапазон целых чисел в дополнительном коде, которые можно передавать по линиям данных?
10. Предположим, что имеется 16 линий адреса и 8 линий данных:
    а) определите адресное пространство памяти, если адресные пространства памяти и ввода-вывода различны;
    б) каков диапазон целых чисел в дополнительном коде, которые можно передавать по линиям данных?
11. Согласно закону Мура, вычислительная мощность компьютеров удваивается каждые 18 месяцев. Каждая следующая версия продуктов корпорации Microsoft работает в полтора раза медленней предыдущей. С

какой скоростью корпорация Microsoft должна выпускать новые версии, чтобы пользователи не заметили действия закона Мура?

## ГЛАВА 2
# ПРЕДСТАВЛЕНИЕ ДАННЫХ

Для написания программ на языке ассемблера Вам придется использовать не только десятичную, но также двоичную и шестнадцатеричную систему счисления.

Всякая электронная вычислительная машина имеет дело с числами и вычислениями, поэтому в первую очередь возникает вопрос, как представить числа в каком-то физическом виде. В первых электронных вычислительных машинах это делалось путем изменения напряжения, и чем выше было напряжение, тем большему числу оно соответствовало. Такой принцип устройства вычислительных машин, называемый аналоговым, оказался малоудобным и ненадежным и вскоре на смену таким машинам пришли цифровые ЭВМ. В первых быстродействующих вычислительных машинах, созданных в США в начале сороковых годов XX века, использовалась десятичная арифметика. Но в 1946 г. в сыгравшем важную роль в развитии вычислительной технике отчете Артура У. Беркса (A.W. Burks), Германа Г. Гольдстайна (H.H. Goldstine) и Джона фон Неймана (J.von Neumann) о проекте первой вычислительной машины с хранимой в памяти программой были подробно изложены причины, которые обосновывали преимущества вычислителей с системой счисления по основанию 2. В основу этого доклада были положены не только теоретические обоснования, но и результаты работ, проводившихся со второй половины 30-х годов Джоном В. Атанасовым (J.V. Atanasoff) и Джорджем Р. Штибитцем (G.R. Stibitz) в США, Л. Куффигналом (L. Couffignal) и Р. Валтой (R. Valtat) во Франции, Гельмутом Шрейером (H. Schreyer) и Конардом Цузе (K. Zuze) в Германии по разработке первых электромеханических и электронных машин для выполнения арифметических операций с числами в двоичной системе счисления. После 12 лет работы с двоичными вычислительными машинами в статье В. Буххольца (W. Buchholz) «Fingers or First?» был выполнен анализ сравнительных достоинств и недостатков двоичной системы счисления. С тех пор вычислительная техника строится на схемах, которые находятся либо в одном состоянии, либо в другом – третьего не дано. Считается, что одно состояние соответствует «логическому нулю», а другое – «логической единице». Использование таких бистабильных схем (*триггеров*) для представления и хранения чисел заставляет перейти к системе, где счет идет двойками, а не десятками, как мы привыкли.

*Бит* (англ. *bit* – кусочек), распространенное на Вест-Индских островах название части разрубленной серебряной или золотой монеты. В XVIII веке это название перешло на мелкие испанские серебряные монеты. Пираты, чтобы расплатиться за мелкие услуги, когда у них не было мелких денег, разрубали дублон или пиастр на 8 частей и одна восьмая монеты называлась *битом*. Представим, что части монеты пронумерованы (рис. 2.1) и разрублено 8 монет.

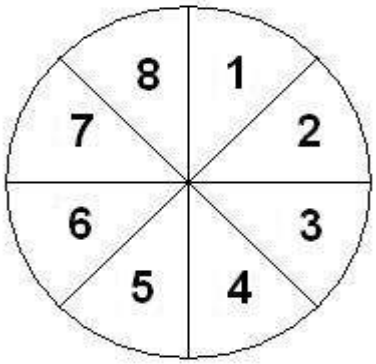



***Рис. 2.1***

Посчитайте сколько вариантов монеты (11111111, 11111112, ..., 88888888) можно собрать из таких битов? Число размещений с повторениями из $n$ элементов по $m$ равно $n^m$. Если не учитывать симметрию, тогда ответ: $8^8$=16777216! Пусть одна половина монеты золотая (1), а другая серебряная (0) (рис. 2.2).

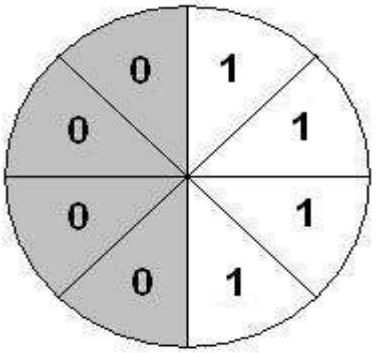



***Рис. 2.2***

Посчитайте сколько вариантов такой монеты (00000000, 00000001, ..., 11111111) можно собрать из битов в этом случае? Ответ: $2^8$=256. Хорошенько запомните это число – оно не один раз встретится вам в этой книге!

В 1948 г. американский математик Джон Уайлдер Таки (John Wilder Tukey) заменил словосочетание *binary digit* (двоичное число) сначала на более короткое *bigit*, а затем на еще более короткое *bit*. Теперь *битом* называют один разряд двоичного кода (числа с основанием 2). Бит – основной структурный блок, из которого строится информация. Один бит информации – это минимально возможное ее количество. Все, что меньше бита, информации не содержит. Бит может принимать только два взаимоисключающих значения: да/нет, 1/0, включено/выключено, черное/белое. Поскольку бит представляет собой минимальный объем, более сложную информацию можно передавать в нескольких битах.

## 2.1. Позиционные системы счисления

Числа в позиционных системах счисления с основанием β определяются следующим образом:

$$\alpha_n\alpha_{n-1}\ldots\alpha_2\alpha_1\alpha_0,\alpha_{-1}\ldots\alpha_k =$$
$$=\alpha_n\beta^n + \alpha_{n-1}\beta^{n-1} + \ldots +\alpha_2\beta^2 + \alpha_1\beta^1 + \alpha_0\beta^0 + \alpha_{-1}\beta^{-1} + \ldots +\alpha_k\beta^{-k} \qquad (1)$$

где α (альфа) – допустимая в данной системе цифра, β (бета) – основание системы счисления, а показатели степени обозначают порядковый номер. Для традиционной десятичной системы возможными значениями α будут цифры от 0 до 9, индекс при α покажет порядок разрядности, β равняется 10, а показатель степени – любое число от плюс бесконечности до минус бесконечности. Между $\alpha_0$ и $\alpha_{-1}$ ставят *разделяющую запятую,* отделяющую целую от дробной части числа. Цифру $\alpha_k$ с бóльшим $k$ называют более значимой, чем $\alpha_k$ с меньшим $k$; крайнюю слева, или «ведущую», цифру называют *наиболее значимой*, а крайнюю справа, или «хвостовую», – *наименее значимой*. Посмотрим на то, как представить по формуле 1 число 987654,321:

$$9_5\times10^5+8_4\times10^4+7_3\times10^3+6_2\times10^2+5_1\times10^1+4_0\times10^0+3_{-1}\times10^{-1}+2_{-2}\times10^{-2}+1_{-3}\times10^{-3}=$$
$$=(9\times100000)+(8\times10000)+(7\times1000)+(6\times100)+(5\times10)+(4\times1)+$$
$$+(3\times0,1)+(2\times0,01)+(1\times0,001).$$

В результате сложения промежуточных сумм получаем число 987654,321.

В каждом формате представления чисел, или *системе счисления*, используется свое *основание* – максимальное значение числа, которое можно представить с помощью одного разряда. В общем случае в качестве β берется целое число, большее 1, а в качестве $\alpha_k$ – целые числа из интервала $0 \leqslant \alpha_k < beta.$ Таким образом приходим к стандартной двоичной (β=2), троичной (β=3), четверичной (β=4), … системам счисления. В таблице 2.1.1 приведены все возможные значения разрядов α в разных системах счисления.

Компьютеры работали, работают и в ближайшее время будут работать в двоичной системе. Человечество же в своем развитии остановилось на десятичной системе счисления – она была выбрана опытным путем в процессе долгого перебора пальцев. Кроме десятичной (*decimal*), при написании программ на языке ассемблер используют двоичную (*binary*), восьмеричную (*octal*), шестнадцатеричную (*hexadecimal*) системы счисления.

*Таблица 2.1.1*

**Цифры разных систем счисления**

| Система счисления | Основание | Используемые цифры |
|---|---|---|
| двоичная | 2 | 0,1 |
| четверичная | 4 | 0, 1, 2, 3 |
| восьмеричная | 8 | 0, 1, 2, 3, 4, 5, 6, 7 |
| десятичная | 10 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
| шестнадцатеричная | 16 | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F |

*Таблица 2.1.2*

**Эквиваленты чисел в разных системах счисления**

| Двоичная | Четверичная | Восьмеричная | Десятичная | Шестнадцатеричная |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 10 | 2 | 2 | 2 | 2 |
| 11 | 3 | 3 | 3 | 3 |
| 100 | 10 | 4 | 4 | 4 |
| 101 | 11 | 5 | 5 | 5 |
| 110 | 12 | 6 | 6 | 6 |
| 111 | 13 | 7 | 7 | 7 |
| 1000 | 20 | 10 | 8 | 8 |
| 1001 | 21 | 11 | 9 | 9 |
| 1010 | 22 | 12 | 10 | A |
| 1011 | 23 | 13 | 11 | B |
| 1100 | 30 | 14 | 12 | C |
| 1101 | 31 | 15 | 13 | D |
| 1110 | 32 | 16 | 14 | E |
| 1111 | 33 | 17 | 15 | F |
| 10000 | 100 | 20 | 16 | 10 |

Вот задача, которая может показаться на первый взгляд абсурдной: чему равно 84, если 8×8=54?

Этот странный вопрос не лишен смысла и задача решается с помощью системы уравнений. Наверное вы догадались, что числа входящие в задачу, написаны не в десятичной системе, иначе вопрос *чему равно 84* выглядит нелепым. Пусть основание неизвестной системы равно $x$. Число 84 означает 8 единиц второго разряда и 4 единицы первого, то есть $84=8x+4$.

Число 54 означает $5x+4$ получили уравнение $8\times8=5x+4$, то есть в десятичной системе $64=5x+4$, откуда $x=12$. Числа написаны по двенадцатеричной системе, и $84_{12}=8\times12+4=100_{10}$. Значит: если $8\times8=54_x$, то $x=12$ и $84_x=100_{10}$.

Подобным же образом решается и другая задача: чему равно 100, когда 5×6=33? Ответ: 81 в девятеричной системе счисления.

Чтобы не запутаться в системах счисления, в конце числа ставят букву *спецификатор*: после двоичного числа букву b (**b**inary), после восьмеричного – o (**o**ctal), после шестнадцатеричного – h (**h**exadecimal), а после десятичного спецификатор, как правило, или не ставится, или ставится буква d (**d**ecimal).

Так как сложно определить, являются ли наборы символов типа BEEF или DEADC0DE переменными или шестнадцатеричными (далее по тексту *hex*) числами, приходится соблюдать следующие правила при их записи:

1. если число начинается с «символа» (A-F), то в начале числа ставят ноль;
2. название переменной не может начинаться с десятичной цифры;
3. хотя при записи шестнадцатеричного числа большие и малые одноименные буквы считаются эквивалентными, но цифры записывают большими буквами, а спецификатор маленькой.

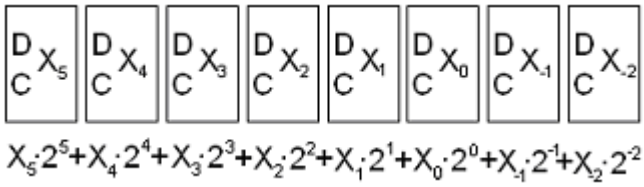



***Рисунок 2.1.1***

Числа, используемые в компьютере, могут быть представлены с помощью таких устройств памяти как триггеры. На рисунке 2.1.1 показан регистр, состоящий из восьми триггеров: $X_5$, $X_4$, $X_3$, $X_2$, $X_1$, $X_0$, $X_{-1}$ и $X_{-2}$, используемых для хранения числа. Пусть $X_5=1$, $X_4=0$, $X_3=1$, $X_2=0$, $X_1=1$, $X_0=1$, $X_{-1}=0$ и $X_{-2}=1$, тогда запись состояния триггеров соответствует двоичному числу 101011,01b или десятичному числу 43,25.

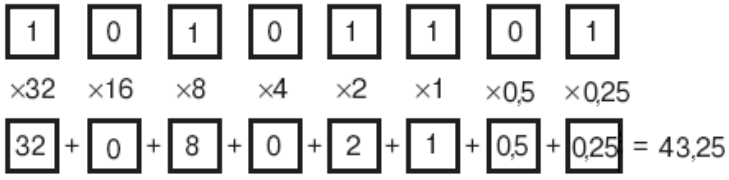



***Рисунок 2.1.2***

### 2.1.1. Шестнадцатеричные числа

Когда мы пишем число 123 в десятичной системе счисления, то подразумеваем:

1 раз по 100 ($100=10^2$) + 2 раза по 10 + 3 раза по 1

Если мы используем число 123h в шестнадцатеричной системе счисления, то подразумевается: 1 раз по 256 ($256=16^2$) + 2 раза по 16 + 3 раза по 1 или десятичное 291:

$123h \rightarrow 291$

Шестнадцатеричное число 5BCh это:

5 раз по 256 + 11 раз по 16 (B=11 по таблице 0) + 12 раз по 1 (C=12)

А теперь переведем 5BCh из шестнадцатеричной в десятичную систему счисления:

$5\times16^2 + 11\times16^1 + 12\times16^0 = 1468 \qquad 5BCh \rightarrow 1468$

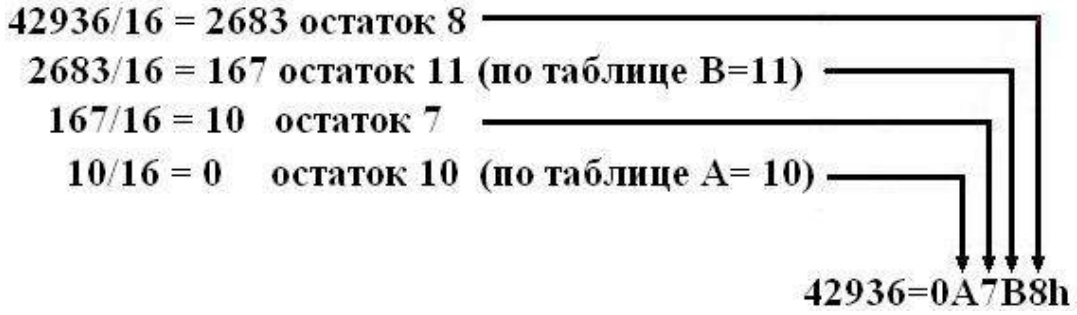
*Второй способ* перевода из шестнадцатеричной в десятичную систему сложения заключается в умножении на 6 на *k*-ом шаге *k* ведущих разрядов и сложении полученных (*k*+1) ведущих разрядов, перед сложением *k* ведущих разрядов числа переводятся в *псевдодесятичный формат* (*пдф*). Пример перевода числа 7Bh в *пдф*: 7×10+11=81 (B=11). Для лучшего понимания внимательно рассмотрите примеры перевода шестнадцатеричных чисел 0ABCh и 0CDEFh в десятичные 2748 и 52719:

```
 +ABC A=10 10×6=60
  60
+111C←переводим AB в ПДФ
  60
+171C  171×6=1026
 1026
+1722←переводим 171C в ПДФ
 1026
 2748
```

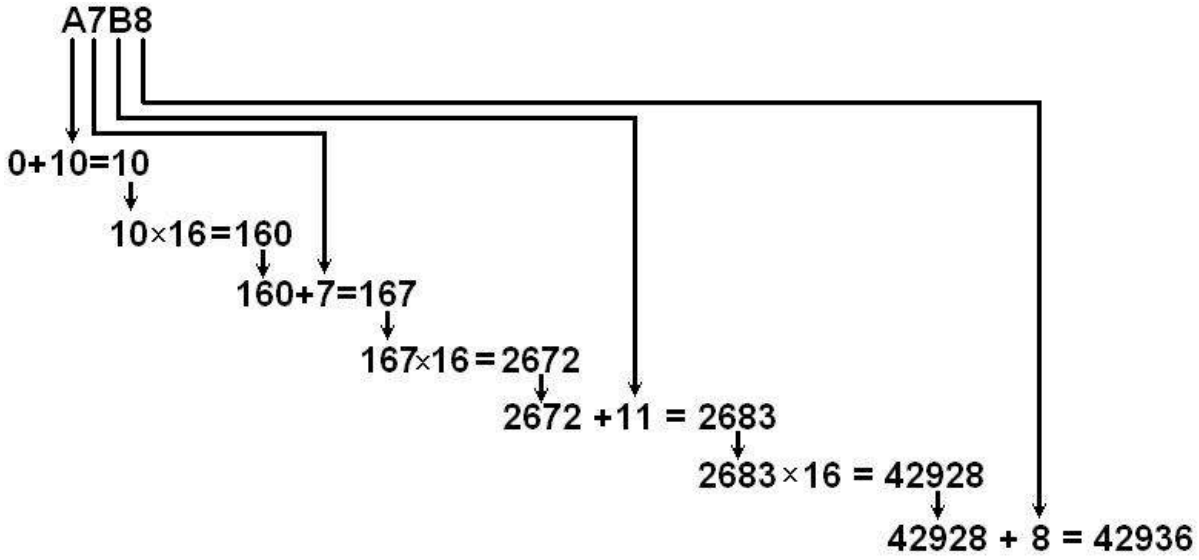
```
 +CDEF C=12 12×6=72
  72
+133EF←переводим CD в ПДФ
  72
+205EF 205×6=1230
 1230
+2064F←переводим 205E в ПДФ
 1230
+3294F 3294×6=19764
 19764
+32955←переводим 3294F в ПДФ
 19764
 52719
```

*Первый способ* преобразование числа из десятичной системы в шестнадцатеричную – сперва находим *последнюю* цифру, затем предпоследнюю и так далее...

Число 42936 переводится следующим образом:



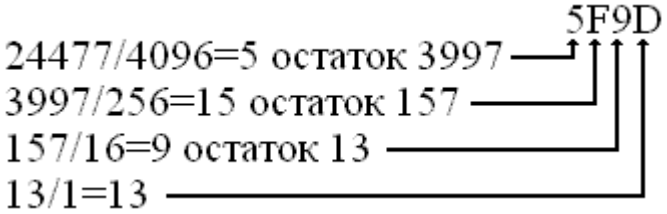

Обратный процесс – переведем шестнадцатеричное число в десятичное – возьмем число 0A7B8h учитывая, что каждая шестнадцатеричная цифра всегда в 16 раз больше ближайшей цифры справа:




0A7B8h → 42936

*Второй вариант* перевода десятичного числа в шестнадцатеричное – сперва находим *первую* цифру, затем вторую и так далее... Для начала придется запомнить степени 16 ($16^5$=1048576, $16^4$=65536, $16^3$=4096, $16^2$=256).

Начинаем с вычисления старшего разряда и определения его порядка в шестнадцатеричном числе, если десятичное число между 1 и 16 миллионами – начинаем с деления на 1048576, если между миллионом и 65 тысячами – начнем с деления на 65536 и так далее. Определившись со степенью $n$ – делим десятичное число на $16^n$. Результатом будет первая шестнадцатеричная цифра и остаток. Остаток делится на $16^{n-1}$ и так далее, пока остаток не станет меньше 16, а степень нулевой. Для примера подсчитаем шестнадцатеричный эквивалент числа 24477:



Для контроля переведем найденное шестнадцатеричное число обратно в десятичное по формуле 1:

$$5F9Dh=5\times4096+15\times256+9\times16+13=20480+3840+144+13=24477$$

### 2.1.2. Двоичные числа

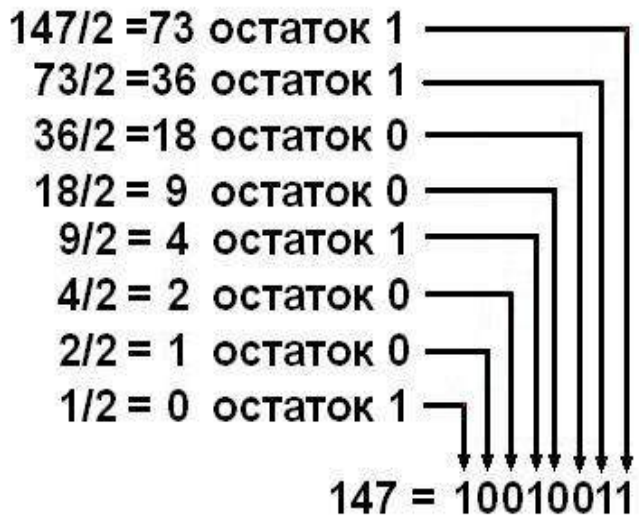
Компьютер работает с двоичной информацией, а человеку удобно производить вычисления, используя десятичную систему счисления. Для перевода из десятичной системы в двоичную можно использовать три способа.

*Способ первый – классический*: ищем двоичный эквивалент десятичного числа от *последней* двоичной цифры к первой. Десятичное число последовательно делится на 2 до получения частного равного нулю. Эквивалент числа в двоичной системе – это упорядоченная последовательность остатков от деления в порядке, обратном их получению.

Делим число на 2. Запишем частное $q$ и остаток $a$.

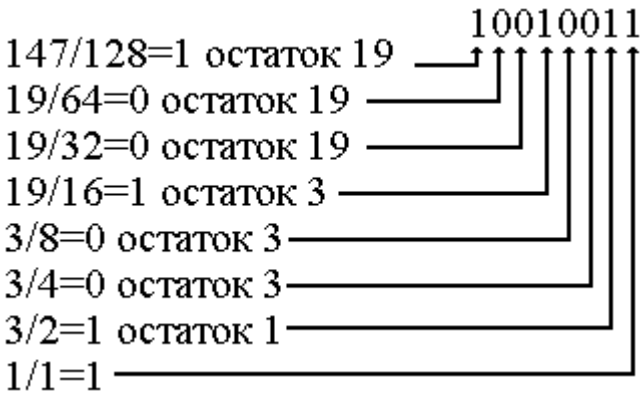
1. Если в результате деления частное $q \neq 0$, то принимаем его за новое делимое и возвращаемся к пункту 1.
2. Если в результате деления частное $q = 0$ выписываем остатки в обратном порядке. Это и будет двоичный эквивалент исходного числа.

Переведем в двоичный эквивалент десятичное число 147:




$$147\rightarrow10010011b$$

*Способ второй* – ищем двоичный эквивалент десятичного числа начиная с *первой* цифры. Для этого придется запомнить степени 2 ($2^{10}$=1024, $2^9$=512, $2^8$=256, $2^7$=128, $2^6$=64, $2^5$=32, $2^4$=16, $2^3$=8, $2^2$=4). Для примера подсчитаем двоичный эквивалент того же числа 147, $2^7=128<147<2^8=256$:




Результат: 147 → 10010011b.

*Способ третий – студенческий*: щелкаем по клавише «Пуск», открываем стандартные приложения, запускаем калькулятор в режиме инженерный, набираем десятичное число и нажимаем на клавишу F8 или мышкой по переключателю с надписью Bin. На экране калькулятора появится двоичный эквивалент числа (рис. 2.1.3).

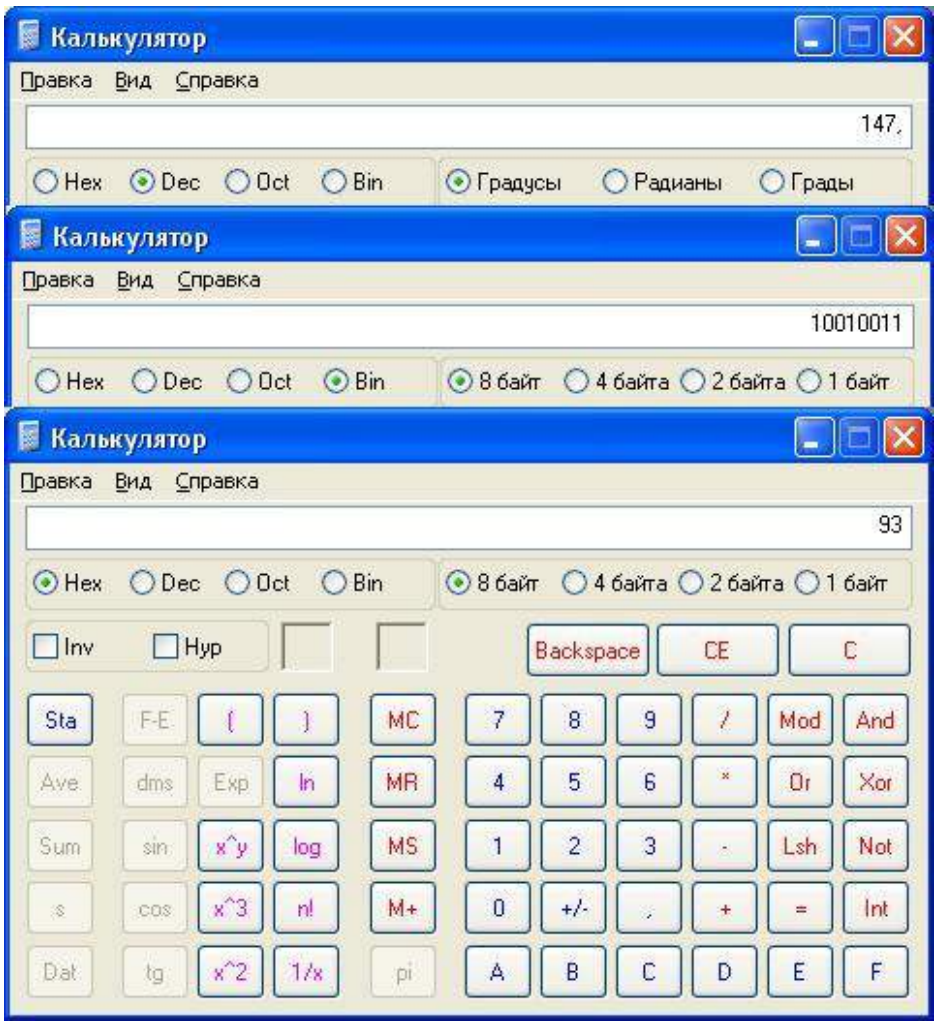



***Рис. 2.1.3 Представление числа 147 в десятичном (Dec), двоичном (Bin) и шестнадцатеричном виде (Hex)***

Для обратного преобразования из двоичной системы счисления в десятичную представьте двоичное число 10100111b в следующем виде

$$1_7\times2^7+0_6\times2^6+1_5\times2^5+0_4\times2^4+0_3\times2^3+1_2\times2^2+1_1\times2^1+1_0\times2^0=128+32+4+2=167$$

или используйте калькулятор.

Двоичные числа могут представлять собой «многометровые» последовательности 0 и 1, а так как при запоминании или записи такого числа легко ошибиться – применяют следующий мнемонический прием – цифры в двоичном числе объединяют в группу по 3 или 4 цифры и запоминают уже эту итоговую цифру. То же самое число 11011b, если добавить недостающие нули можно представить и как 011.011b или 33o (восьмеричное число), и как 0001.1011b или 1Bh (шестнадцатеричное число). Докажем это. Пусть у нас есть двоичное число $N$ с восемью разрядами до и четырьмя разрядами после запятой:

$$N = b_7b_6b_5b_4b_3b_2b_1b_0,b_{-1}b_{-2}b_{-3}b_{-4}$$

В двоичной системе это число равно:

$$N=b_7\times2^7+b_6\times2^6+b_5\times2^5+b_4\times2^4+b_3\times2^3+b_2\times2^2+b_1\times2^1+b_0\times2^0+b_{-1}\times2^{-1}+b_{-2}\times2^{-2}+$$
$$+b_{-3}\times2^{-3}+b_{-4}\times2^{-4}=(b_7\times2^3+b_6\times2^2+b_5\times2^1+b_4\times2^0)\times2^4+$$
$$+(b_3\times2^3+b_2\times2^2+b_1\times2^1+b_0\times2^0)\times2^0+(b_{-1}\times2^3+b_{-2}\times2^2+b_{-3}\times2^1+b_{-4}\times2^0)\times2^{-4}=$$
$$=a_1\times2^4+a_0\times2^0+a_{-1}\times2^{-4}=a_1\times16^1+a_0\times16^0+a_{-1}\times16^{-1},$$

где $a_1=b_7\times2^3+b_6\times2^2+b_5\times2^1+b_4\times2^0$

$a_0=b_3\times2^3+b_2\times2^2+b_1\times2^1+b_0\times2^0$

$a_{-1}=b_{-1}\times2^3+b_{-2}\times2^2+b_{-3}\times2^1+b_{-4}\times2^0$ .

Из представления $N=a_1\times16^1+a_0\times16^0+a_{-1}\times16^{-1}$ следует, что шестнадцатеричными цифрами числа $N$ являются $a_1$, $a_0$, $a_{-1}$, а равенство $a_1=b_7\times2^3+b_6\times2^2+b_5\times2^1+b_4\times2^0$ показывает – если записать шестнадцатеричную цифру $a_1$ в виде четверки двоичных цифр, то этими двоичными цифрами будут $b_7$, $b_6$, $b_5$, $b_4$. Поскольку равенства аналогичны и для $a_0$, $a_{-1}$, тем самым доказано, что двоичная запись числа $N$ совпадает с его шестнадцатеричной записью.

Для перевода целого двоичного числа в шестнадцатеричное число его разряды, начиная с крайнего правого, группируются по четыре, недостающие цифры слева заменяются нулями, а затем каждая группа заменяется согласно таблице на соответствующий двоичный эквивалент. Например, число 110.0100.1011.0101b в шестнадцатеричном виде представляется числом 64B5h:

| | | | | |
|---|---|---|---|---|
| bin: | 0110 | 0100 | 1011 | 0101 |
| hex: | 6 | 4 | B | 5 |

Для перевода шестнадцатеричного числа в двоичное число каждая

цифра шестнадцатеричного числа заменяется, согласно таблице 2.1.2 на соответствующее двоичное значение.

| hex: | 7 | F | 0 | E |
|---|---|---|---|---|
| bin: | 0111 | 1111 | 0000 | 1110 |

Представление числа в двоичной, десятичной, восьмеричной или в любой другой системе представления чисел – это личное дело каждого, но вот компьютер оперирует исключительно с двоичными числами, то есть с битами. Совокупность двоичных разрядов, выражающих числовые или иные данные, образует некий битовый рисунок (таблица 2.1.3). Практика показывает, что с битовым представлением удобнее работать, если этот рисунок имеет регулярную форму. В настоящее время в качестве таких форм используется последовательность из восьми *взаимосвязанных* битов, которые называются *байтами* (BYTE – **BINARY TERM**). Биты называются взаимосвязанными потому, что для значения байта важно не только сколько битов в нем включено (равно 1), но и их местоположение. Значение байта определяется по правилам позиционной арифметики, то есть чем левее расположены включенные биты, тем больше значение байта: 10000000b>011111111b. Нумерация битов в байте, слове или иной конструкции будет всегда начинаться с 0 и вестись слева на право.

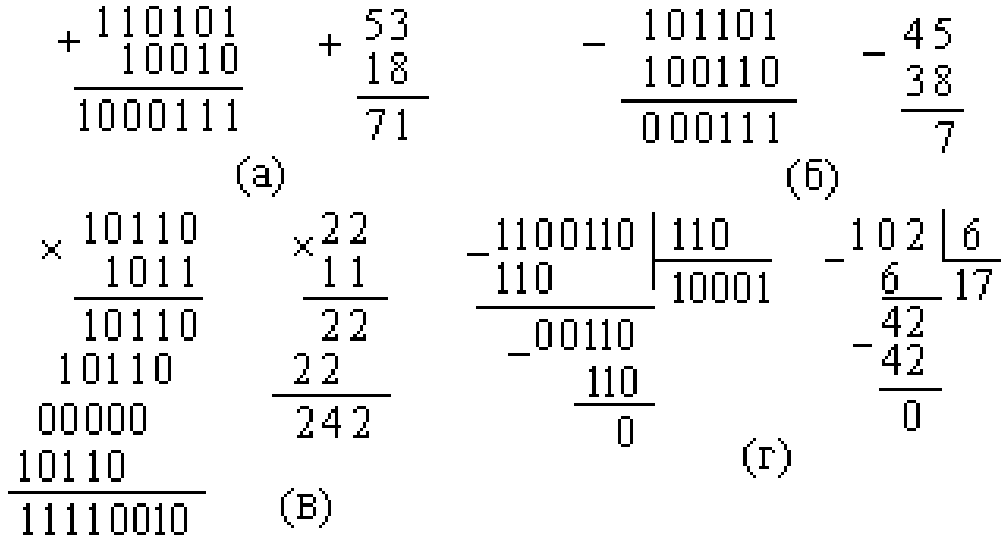



***Рис. 2.1.4. Арифметические операции в двоичном и десятичном видах***

Арифметические операции в любой системе счисления выполняются по тем же алгоритмам, что и в десятичной системе. На рисунке 2.1.4 (а) показано сложение, (б) – вычитание, (в) – умножение и (г) –деление.

Во многих случаях целесообразно использовать не 8-битное кодирование, а 16-битное, 24-битное, 32-битное и более.

*Таблица 2.1.3*

| Десятичное число | Hex | Bin | байт |
|---|---|---|---|
| 0 | 0 | 0 | 0000 0000 |
| 1 | 1 | 1 | 0000 0001 |
| 2 | 2 | 10b | 0000 0010 |
| 3 | 3 | 11b | 0000 0011 |
| 4 | 4 | 100b | 0000 0100 |
| … | ... | … | … |
| 255 | 0FFh | 11111111b | 1111 1111 |

### 2.1.3. Почему в байте именно 8 бит?

Ранние ЭВМ имели размеры машинных слов и байтов, отличные от 8 бит, как правило, кратные шести. В 1950-х–1960-х годах многие ЭВМ использовали 6-битную кодировку символов, поэтому длина байта была первоначально кратна шести битам. Восемь бит в байте использовалось в IBM System/360. Это стало стандартом де-факто. С начала 1970-х в вычислительной технике используют байты, состоящие из 8 бит, и машинные слова, кратные 8 битам. Появление 8-битных байтов у System/360 связано, вероятно, с использованием BCD-форматом представления числа (BCD – *binary-coded decimal – двоично-десятичный код*): по 4 бита на каждую цифру (0-9), таким образом один байт представлял две десятичные цифры. В System/360 были специальные инструкции для обработки данных такого формата, для представления BCD было бы трудно использовать 6-битные байты, поэтому 8 бит в байте стало наилучшим решением. По другой версии, 8-битный размер байта связан с 8-битным числовым представлением символов в кодировке EBCDIC: один байт – один символ. Остается удивляться прозорливости пиратов, которые в XVII назвали битом *именно* $\frac{1}{8}$ -ю монеты!

Группа из 16 взаимосвязанных бит (двух взаимосвязанных байт) называется *машинным словом* (WORD). Соответственно группа из четырех взаимосвязанных байтов (32 бита или два слова) называется *удвоенным словом* (DOUBLE WORD). Группа из восьми байтов (64 бита или два удвоенных слова) – *учетверенным словом* (QUADRUPLE WORD), группа из 16 байтов (128 бит или два учетверенных слова) – *параграфом* (PARAGRAPH) или *двойным учетверенным словом* (DOUBLE QUADRUPLE WORD), группа из 4096 байтов (256 параграфов) – *страницей* (PAGE) (таблица 2.1.4).

Термины «бит», «байт» и «слово» используются для описания как элементов данных, которые обрабатывает компьютер, так и элементов памяти.

*Таблица 2.1.4*

**Типы данных**

| Тип в ассемблере | | Сокращение | Соответствие в ЯВУ | | Количество | |
|---|---|---|---|---|---|---|
| | | | C/C++ | Pascal/Delphi | байт | бит |
| байт | byte | b | char | Byte (Shortint) | 1 | 8 |
| слово | word | w | short | Word (Integer) | 2 | 16 |
| двойное слово | dword | d | int | Longint | 4 | 32 |
| учетверенное слово | qword | q | long | – | 8 | 64 |
| двойное учетверенное слово | dqword | dq | Long long | – | 16 | 128 |
| параграф | paragraph | para | – | – | 16 | 128 |
| страница | page | page | – | – | 4096 | 32768 |

## 2.2. Представление отрицательных двоичных целых чисел

В общем случае под целое число можно отвести любое число соседних битов памяти, однако система команд компьютера поддерживает работу с числами размером в байт, слово, двойное слово. Поэтому целые числа представляются только байтом, словом, двойным словом и так далее.

Рано или поздно возникает потребность представлять в двоичном коде отрицательные числа. А как в двоичной системе представить отрицательное число, если мы не можем использовать для этой цели минус как в традиционной десятичной системе? Как, например, должно выглядеть в двоичной системе –1?

По определению, для каждого натурального числа $n$ существует одно и только одно отрицательное число, обозначаемое $-n$, которое дополняет $n$ до нуля:

$$n + (-n) = 0.$$

Найдем такое 4-битное число, которое при сложении с числом 5 (0101b) дало бы нам ноль, переносом в 5-й бит мы пренебрегаем (рис 2.2.1).

```
  0 1 0 1
+ 1 0 1 1
---------
  0 0 0 0
```

*Рис. 2.2.1*

Теперь самостоятельно заполните таблицу 2.2.1.

*Таблица 2.2.1*

| **hex-число** | **двоичное число *n*** | **двоичное число *–n*** |
|---|---|---|
| 0 | 0000 | |
| 1 | 0001 | |
| 2 | 0010 | |
| 3 | 0011 | |
| 4 | 0100 | |
| 5 | 0101 | 1011 |
| 6 | 0110 | |
| 7 | 0111 | |
| 8 | 1000 | |
| 9 | 1001 | |
| A | 1010 | |
| B | 1011 | 0101 |
| C | 1100 | |
| D | 1101 | |
| E | 1110 | |
| F | 1111 | |

Так как $n = -(-n)$, поэтому, таблицу 2.2.1 можно сократить в два раза

| **число *n*** | | **число *–n*** | |
|---|---|---|---|
| **Hex** | **bin** | **Hex** | **bin** |
| 0 | 0000 | 0 | 0000 |
| 1 | 0001 | F | 1111 |
| 2 | 0010 | E | 1110 |
| 3 | 0011 | D | 1101 |
| 4 | 0100 | C | 1100 |
| 5 | 0101 | B | 1011 |
| 6 | 0110 | A | 1010 |
| 7 | 0111 | 9 | 1001 |
| 8 | 1000 | 8 | 1000 |

При этом оказалось, что числа 0 и 8 являются сами для себя отрицательными числами. Если не рассматривать числа 0 и 8, числа в левой половине таблицы объединяет то, что самый старший (четвертый) бит у них равен 0, а у чисел в правой половине самый старший бит равен 1. Пусть са-

мый старший бит считается знаковым. Если он равен 0, число будем считать положительным, если 1 – отрицательным. Мы только что изобрели способ представления положительных и отрицательных чисел в двоичной системе, так называемый *код дополнения до двух,* или *дополнительный код.* При этом, число 0 может быть только положительным, а четырех битовое число 8 – только отрицательным.

| **Положительное 4-битное число *n*** | | | **Отрицательное 4-битное число *–n*** | | |
|---|---|---|---|---|---|
| **десятичное** | **hex** | **bin** | **десятичное** | **hex** | **bin** |
| +0 | 0 | **0**000 | **–**0 | – | – |
| +1 | 1 | **0**001 | **–**1 | F | **1**111 |
| +2 | 2 | **0**010 | **–**2 | E | **1**110 |
| +3 | 3 | **0**011 | **–**3 | D | **1**101 |
| +4 | 4 | **0**100 | **–**4 | C | **1**100 |
| +5 | 5 | **0**101 | **–**5 | B | **1**011 |
| +6 | 6 | **0**110 | **–**6 | A | **1**010 |
| +7 | 7 | **0**111 | **–**7 | 9 | **1**001 |
| +8 | – | – | **–**8 | 8 | **1**000 |

Компьютер различает целые числа без знака (неотрицательные) и целые числа со знаком (таблица 2.2.2).

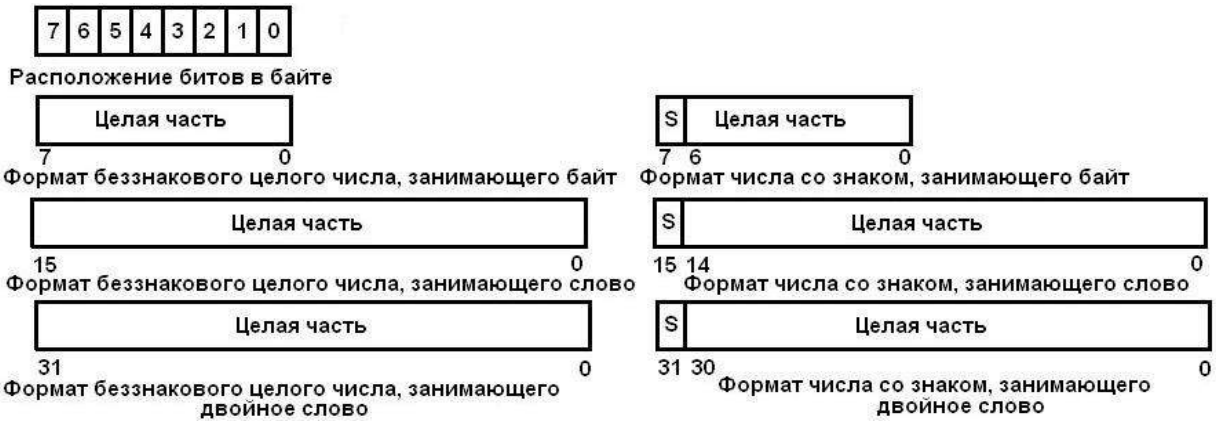



***Рис. 2.2.2. Форматы представления целых чисел***

*Таблица 2.2.2*

**Дополнительный код**

| дополнительный двоичный код | Hex | число со знаком | число без знака |
|---|---|---|---|
| 0000 0000 | 00h | 0 | 0 |
| 0000 0001 | 01h | 1 | 1 |
| 0000 0010 | 02h | 2 | 2 |
| | ... | | |
| 0111 1111 | 7Fh | 127 | 127 |
| 1000 0000 | 80h | –128 | 128 |
| 1000 0001 | 81h | –127 | 129 |
| | ... | | |
| 1111 1110 | 0FEh | –2 | 254 |
| 1111 1111 | 0FFh | –1 | 255 |

Для целых чисел со знаком положительное число записывается в прямом коде (как есть). Отрицательное – в дополнительном коде. Преобразовать положительное число в отрицательное можно тремя способами.

*Способ первый – классический*: взять число в прямом коде, преобразовать его в *инверсный* код. Для этого все нули этого числа необходимо заменить на единицы, а единицы на нули. К числу в инверсном коде добавить 1. На рис. 2.2.3 пример преобразования числа +7 в –7.

```
+7=  0000 0111   прямой код
     1111 1000   инверсный код
   +  0000 0001
–7=  1111 1001   дополнительный
                      код
```

***Рис. 2.2.3***

*Способ второй – практический*: возьмем максимальное число (для байта $2^8$=256, для слова $2^{16}$=65536, для двойного слова $2^{32}$=4294967296) и отнимем от него наше число, получившееся число преобразуем в двоичный или **hex-**код:

$$256 – 7 = 249 = 1111\ 1001b = 0F9h = – 7.$$

*Способ третий – студенческий*: щелкаем по клавише «Пуск», открываем стандартные приложения, запускаем калькулятор в режиме инженерный, теперь щелкаем по клавише 7 и «+/–» на экранчике калькуля-

тора появится –7. А теперь жмем на клавишу F8 или мышкой по переключателю с надписью «Bin». Что мы видим на экране калькулятора? Число 11111111111111111111111111111001b. Многовато, на экране – двойное слово. Жмем на F4 или мышью по переключателю с надписью Byte. Наше число приобрело нормальный вид 11111001b.

На самом деле, нет необходимости пересчитывать отрицательные числа. Вы пишете –7, а транслятор автоматически подставит туда, где нужно число 11111001b.

Если мы пренебрегаем знаковым разрядом, то диапазон чисел увеличивается в 2 раза:

$$2^7=128;\ 2^8=256.$$

Попрактикуемся и найдем 8-битный эквивалент –5:

| | |
|---|---|
| 0000 0101b | двоичное представление +5. |
| 1111 1010b | инвертируем все биты. |
| 1111 1011b | добавляем 1. |

Проделаем обратное преобразование:

| | |
|---|---|
| 1111 1011b | двоичное представление –5. |
| 0000 0100b | инвертируем все биты. |
| 0000 0101b | добавляем 1 и получаем +5. |

А теперь поэкспериментируем с 16-битными положительными и отрицательными числами: 7FFFh (+32767, наибольшее 16-битное положительное число), 4000h (+16384), 0 и 8000h (–32768, наименьшее 16-битное отрицательное число).

Получим отрицательные эквиваленты чисел:

| | |
|---|---|
| 0111 1111 1111 1111b | 7FFFh (+32767) |
| 1000 0000 0000 0000b | инвертируем все биты (8000h) |
| 1000 0000 0000 0001b | добавляем 1 (8001h или −32767) |
| | |
| 0100 0000 0000 0000b | 4000h (16384) |
| 1011 1111 1111 1111b | инвертируем все биты (0BFFFh) |
| 1100 0000 0000 0000b | добавляем 1 (0C000h или −16384) |
| | |
| 0000 0000 0000 0000b | 0 |
| 1111 1111 1111 1111b | инвертируем все биты (0FFFFh) |
| 0000 0000 0000 0000b | добавляем 1 (0) |

«Ноль» – он и в Африке «ноль».

Получим положительный эквивалент числа 8000h:

| | |
|---|---|
| 1000 0000 0000 0000b | 8000h (−32768) |
| 0111 1111 1111 1111b | инвертируем все биты (7FFFh) |
| 1000 0000 0000 0000b | добавляем 1 (8000h или −32768). |

После смены знака число 8000h не изменилось! –(–32768)=–32768. Почему? Дело в том, что число +32768 нельзя представить как 16-битное число со знаком (вспомните, то же самое у нас получилось с четырехбитным числом 8). При выполнении операции смены знака у числа –32768 микропроцессор x86 выставит признак арифметического переполнения.

## 2.3. Расширение знака и расширение нуля

Возьмем десятичное число –64. Ему соответствует 8-битное число 0C0h, 16-битное 0FFC0h и 32-битное 0FFFFFFC0h. Для числа +64 8-битный эквивалент 40h, 16-битный – 0040h и 32-битный – 00000040h.

*Таблица 2.3.1*

**Примеры расширения знака**

| 8-битное число | 16-битное число | 32-битное число |
|---|---|---|
| 80h | 0FF80h | 0FFFFFF80h |
| 28h | 0028h | 00000028h |
| 9Ah | 0FF9Ah | 0FFFFFF9Ah |
| 7Fh | 007Fh | 0000007Fh |
| – | 1020h | 00001020h |
| – | 8088h | 0FFFF8088h |

Разницу между 8 и 16/32/64-битными эквивалентами можно выразить следующими словами: «если число отрицательное, то слева к 8-битному эквиваленту дописывается 0FF/0FFFFFF/0FFFFFFFFFFFFFF, а если число положительное, то к 8-битному числу слева дописываются нули» или, другими словами, происходит «растягивание» знакового бита.

Если мы рассматриваем беззнаковое десятичное число 128, то ему соответствует 8-битное число 80h, 16-битное – 0080h и 32-битное – 00000080h. Для расширения 8-битного числа до 16/32/64-битного к нему слева дописывают недостающие нули, происходит «расширение нуля».

*Таблица 2.3.2*

**Примеры расширения нуля**

| 8-битное число | 16-битное число | 32-битное число |
|---|---|---|
| 80h | 0080h | 00000080h |
| 28h | 0028h | 00000028h |
| 9Ah | 009Ah | 0000009Ah |
| 7Fh | 007Fh | 0000007Fh |
| – | 1020h | 00001020h |
| – | 8088h | 00008088h |

Иногда, для сокращения размера кода команд, вместо 32-битного

числа используют его 8-битный эквивалент, но это можно проделать лишь с ограниченным диапазоном чисел от –128 до +127.

0FFFFFF80h (–128) можно сократить до 8 бит (80h)
00000040h (+64) можно сократить до 8 бит (40h)
0FFFFFE40h (–448) нельзя сократить до 8 бит
00000100h (+256) нельзя сократить до 8 бит.

## 2.4. Представление чисел с плавающей запятой

У десятичных чисел каждая позиция числа соответствует степени числа 10, то есть число $1234{,}56=1\times10^3+2\times10^2+3\times10^1+4\times10^0+5\times10^{-1}+6\times10^{-2}$. Запятая показывает границу между позицией, соответствующей $10^0$, и дробной частью. В дробной части позиции числа также являются степенями числа 10, но эти степени теперь отрицательные. Дробные двоичные числа записываются аналогично дробным десятичным, но основание системы счисления здесь 2, а не 10. Например, $1{,}101\text{b}=1+\frac{1}{2}+\frac{1}{8}=1\frac{5}{8}=1{,}625$.

Чтобы получить эквивалент целого числа в двоичной системе, мы использовали последовательное деление на 2. Чтобы получить эквивалент десятичной дроби, используем обратную операцию – последовательное умножение на 2.

Для перевода десятичной дроби в двоичную систему можно использовать три способа, а уж который из них Вам покажется более удобным – выбирайте сами.

*Способ первый.* Дробную часть числа последовательно умножают на 2 пока либо дробная часть не станет равной нулю, либо не будет получено необходимое количество разрядов. Целые значения, получаемые при умножении, будут являться эквивалентом десятичной дроби в двоичной системе. Посмотрите конкретный пример на рис. 2.4.1.

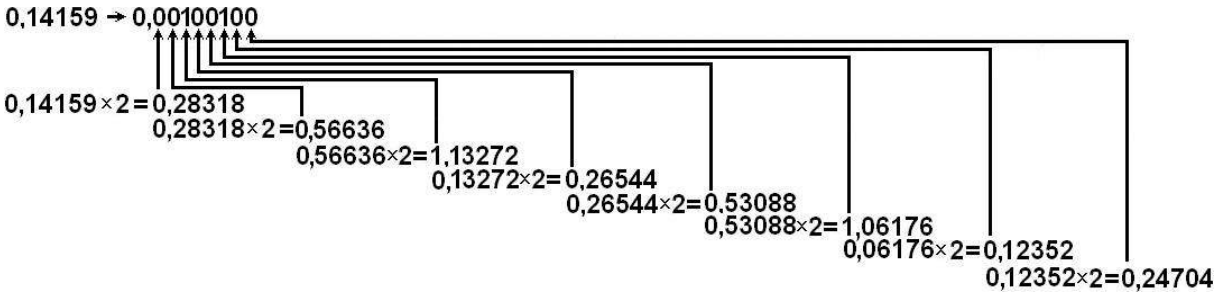



***Рис. 2.4.1***

*Способ второй*. Любое дробное двоичное число можно представить как

$$x_1\times0{,}5+\ x_2\times0{,}25+\ x_3\times0{,}125+\ldots,$$

где $x_1$, $x_2$, …$x$n – нули или единицы, то есть 0,1=0,0001100b.

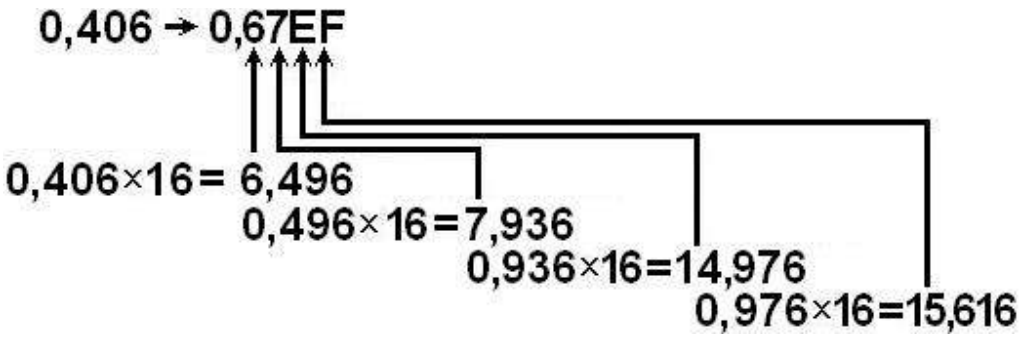
*Способ третий*. Переведем в двоичную систему, например, число

0,27=27/100. Ближайшая к 100 степень двойки $2^7$=128.

$$\frac{27}{100}\times\frac{1,28}{1,28}=\frac{34,56}{128}\approx\frac{35}{128}=\frac{32+2+1}{128}=\frac{1}{4}+\frac{1}{64}+\frac{1}{126}=2^{-2}+2^{-6}+2^{-7}\ =$$

$$=0,01b+0,000001b+0,0000001b=0,0100011b$$

Перевод десятичного дробного числа 0,406 в hex-систему делается аналогично переводу к двоичному виду:




0,616×16=**9**,856 и так далее, то есть 0,406=0,67EF9...h .

Для перевода вещественного целого числа в двоичную или hex-систему необходимо целую и дробную часть перевести по отдельности в двоичную или hex-систему, а затем соединить их. Например, число π≈ 3,14159 должно выглядеть вот так 11,00100100b, или так 3,243Fh.

Существует два способа записи вещественных чисел:

*Первый способ* – целая часть числа отделяется от дробной символом запятой, расположенной между конкретными разрядами в фиксированной позиции. Такой способ записи называют представлением вещественных чисел с *фиксированной запятой*. Если отвести под целую часть числа 8 разрядов, а под дробную 8 разрядов, то максимально большое число, которое можно записать таким способом:

11111111,11111111b=255,99609375≈256 ,

а минимальное возможное число –

00000000,00000001b=1/256≈0,00390625 .

*Второй способ* применяется в астрономии, физике, химии, математике для записи очень больших либо очень маленьких чисел. Любое число представляется в виде $\pm M\times R^P$. Часть числа, представляющую значащие разряды – *M*, называют *мантиссой*. *R* – *основание* системы счисления. *P* – *показатель степени,* определяющий на сколько разрядов вправо или влево необходимо переместить запятую в мантиссе, называется *порядком* числа. Число *M* должно быть $1\leqslant M<R$ . Такой способ записи называют представлением вещественных чисел с *плавающей запятой*. Если под мантиссу отвести 8 разрядов и 8 разрядов под порядок числа (из 8 разрядов 1 разряд отводится под знак и 7 разрядов под число), то максимально возможное

наибольшее число $2^{127} \approx 10^{38}$, а минимальное возможное число $2^{-127} \approx 10^{-38}$.

Для представления в компьютере дробного числа первый разряд числа считается знаковым и обозначает знак мантиссы. За ним следует *мантисса* (число со значением больше или равным 1 и меньшим 2) и *порядок* (степень числа 2). Представление чисел в виде мантиссы и порядка позволяют сводить умножение и деление чисел к сложению и вычитанию показателей степеней, а возведение в степень и извлечение корня – к умножению и делению на показатель степени, что упрощает и сокращает сложные вычисления.

Вещественные числа в памяти компьютера хранятся в нормализованном виде, то есть оно обязательно должно иметь следующий вид:

$$(-1)^S \times M \times 2^P,$$

где значение S определяет знак (S=0 – число положительное, S=1 – отрицательное), M – двоичная мантисса числа, P – порядок двоичного числа.

Из-за того, что старший разряд двоичной мантиссы всегда равен 1, его, из соображений увеличения разрядности числа, предпочитают «хранить в уме».

Для упрощения вычислений значение порядка хранят не в дополнительном, а в смещенном коде (таблица 2.4.1). В смещенном коде число суммируется с константой, которая для *N*-битного кода равна $2^{N-1}-1$.

*Таблица 2.4.1*

| **Десятичное значение** | **Смещенный код** | | |
|---|---|---|---|
| | **8-битный** | **11-битный** | **15-битный** |
| –16383 | – | – | 0000h |
| –1023 | – | 000h | 3C00h |
| –127 | 00h | 380h | 3F80h |
| | ... | | |
| –1 | 7Eh | 3FEh | 3FFEh |
| 0 | 7Fh | 3FFh | 3FFFh |
| 1 | 80h | 400h | 4000h |
| | ... | | |
| 128 | 0FFh | 47Fh | 407Fh |
| 1024 | – | 7FFh | 43FFh |
| 16384 | – | – | 7FFFh |

32/64-разрядные микропроцессоры семейства 80x86 могут работать с 32/64/80-битными вещественными числами (одинарная/двойная/повышенная точность), а также со 128-битными, состоящими из четырех упако-

ванных вещественных чисел одинарной точности (таблица 2.4.2).

*Таблица 2.4.2*

| Количество бит | Тип | Поле знака | Поле порядка | Константа | Поле мантиссы | Диапазон принимаемых значений |
|---|---|---|---|---|---|---|
| 32 | REAL4 | 1 бит | 8 бит | $2^7-1=127$ | 23 бита | От +/– $1{,}18\times10^{-38}$ до $3{,}4\times10^{38}$ |
| 64 | REAL8 | 1 бит | 11 бит | $2^{10}-1=1023$ | 52 бита | От +/– $2{,}23\times10^{-308}$ до $1{,}79\times10^{308}$ |
| 80 | REAL10 | 1 бит | 15 бит | $2^{14}-1=16383$ | 64 бита | От +/–$3{,}37\times10^{-4932}$ до $1{,}18\times10^{4932}$ |

Число 193,75 в формате вещественного числа, занимающего два слова:

$$193{,}75=11000001{,}11=1{,}100000111b\times2^7=4341C000h$$

Преобразуем число 4341C000h в бинарное представление:

| | |
|---|---|
| **0**10000110100000111000000000000000 | Знак числа 0=«+» 1=«–» |
| 0**10000110**10000011100000000000000 | Порядок 86h=+7 в смещенном коде 7+127=134 |
| 01000011**10000011100000000000000** | мантисса числа=(1),1000001110000b |


***Рис. 2.4.2. Форматы представления вещественных чисел***

Запись одного и того же числа в разных представлениях, естественно, будет отличаться. Число 193,75 может быть представлено как:

*Таблица 2.4.3*

| Количество бит | Тип | hex-представление числа 193,75 |
|---|---|---|
| 32 | REAL4 | 4341C000h |
| 64 | REAL8 | 4068380000000000h |
| 80 | REAL10 | 4060C1C0000000000000h |

Преобразуем число 4068380000000000h в бинарное представление:

| |
|---|
| **0**10000000110100000111000000000000000000000000...0000000000000000 |
| Знак числа 0=«+» |
| **0100000001101**0000011100000000000000000000000...0000000000000000 |
| Порядок =+7 7+1023=1030= 406h |
| **0100000001101000001110000000000000000000000000...0000000000000000** |
| мантисса числа=(1),1000001110000000000000000000000000...000000000000b |

Преобразуем число 4060C1C0000000000000h в бинарное представление:

| |
|---|
| **0**10000000110000011000001110000000000000000000000000000000000000...0000 |
| Знак числа 0=«+» |
| **0100000001100000**1100000111000000000000000000000000000000000000...0000 |
| Порядок =+7 7+16383=16480=4060h |
| **0**10000000110000011000001110000000000000000000000000000000000000...0000 |
| мантисса числа=1,1000001110000000000000000000000000000000000...0000b |

Типы REAL4, REAL8 и REAL10 соответствуют стандартным вещественные типам Single (4 байта), Double (8 байта) и Extended (10 байт) в C/C++ и Pascal. Первые два типа (REAL4 и REAL8) являются «упакованными», а REAL10 – «неупакованным» типом, в нем старший, всегда единичный, бит нормализованной мантиссы хранится в явном виде. Все вычисления внутри FPU происходят после автоматического преобразования типов REAL4 и REAL8 в REAL10.

Для записи вещественного числа в программе пересчет вещественного десятичного числа из десятичной системы в hex или двоичную самому делать не обязательно, просто запишите число, используя в качестве разделителя точку вместо запятой:

```
pi dt 3.14159265358979323846264338327795
```

А компилятор сам подставит 32-битный эквивалент в случае с директивой dd, 64-битный для директивы dq или 80-битный для директивы dt.

Теперь читателю предлагается потренироваться в переводе чисел. Пред вами два простых примера: 0.125 и 0.625. Можно усложнить их:

23.125 и 456.625. Проверьте свои ответы, записав результат в переменную типа dword, и посмотрев под отладчиком число в стеке FPU. Автор настаивает на такой практике, даже если вы не новичок.

## 2.5. Битовое поле

Непрерывная последовательность бит, в которой каждый бит является независимым и может рассматриваться как отдельная переменная. Битовое поле может начинаться с любого бита и содержать до 32 бит.

## 2.6. Буквено-цифровые символы

Символы должны декларироваться до их употребления. Описание символа связывает внутренний номер символа с его изображением. Внутри описания номер символа может встречаться только один раз.

Компьютеры обрабатывают не только числа, но и главным образом тексты, поэтому для ввода данных с клавиатуры и для последующего их вывода на принтер или на экран договорились рассматривать байты не как числа, а как буквено-цифровые символы. Для того, чтобы данные можно было передавать между ЭВМ и периферийными устройствами различных производителей договорились использовать один из стандартных кодов для обмена информацией. Кодирование символов – операция в достаточной мере произвольная, и поэтому существует несколько стандартных буквено-цифровых кодов, например 8-разрядный двоично-десятичный код EBCDIC (*Extended Binary Coded Decimal Interchange Code*), применяющийся на мэйнфреймах IBM, код Холлерита, для записи информации на перфокартах, 5-разрядный код Бодо, который долгое время являлся стандартным кодом для телетайпов. Наиболее широко распространенным буквено-цифровым кодом является ASCII (или ANSI). ASCII – аббревиатура от *American Standard Code for Information Interchange* (Американский стандартный код для обмена информацией). Для представления всех букв, цифр и служебных знаков, появляющихся на экране компьютера, обычно используется всего один байт. Всего в ASCII-коде $2^8$=256 символов. ASCII-код используется в мини- и микро-ЭВМ, в том числе и в системах построенных на базе микропроцессоров x86. Символы ASCII включают все символы «американской пишущей машинки» (коды от 32 до 127), специальные «управляющие» коды, такие как табуляция, возврат каретки и так далее (коды от 0 до 31), символы псевдографики и символы для иностранных алфавитов (коды от 128 до 255, так называемая *расширенная таблица* символов) (таблица 2,6,2), позволяющие использовать разные умляуты (Â, Á, Å и т.д.) и рисовать псевдографические рисуночки вроде панелей DOS-овского Нортона и progress bar'ов во время копирования. С кодировкой ASCII конкурируют кодовые страницы от MicroSoft и древнее чудовище

EBCDIC, почти вымершее со времен господства мэйнфреймов IBM. В настоящее время ASCII код морально устарел и постепенно вытесняется Unicode'ом.

При нажатии, например, клавиши «W» на клавиатуре произойдет формирование и передача в компьютер соответствующего этой букве ASCII-кода – 57h. Если же компьютер посылает на монитор или принтер ASCII-код, например 59h, монитор или принтер должны дешифровать эти биты и отреагировать соответствующим образом – вывести на экран или на бумагу букву «Y». ASCII-код используется также для создания файлов в общем формате. Хранение данных в формате ASCII обычно менее компактно, чем в двоичном формате, однако преимущество кода состоит в том, что он является стандартной основой для обмена данными.

Кроме буквенно-цифровых кодов ASCII содержит непечатаемые управляющие символы, частично заимствованные из телетайпных кодов: NULL (нуль) – символ содержащий нули во всех разрядах и применяющийся для разделения цепочки символов, а также для других целей, CR (возврат каретки) и LF (перевод строки), устанавливающиеся при печати новой строки, FF (перевод формата), использующийся для перехода к новой странице, ESC, служащий в качестве разделителя команд, ETX (конец текста или «управляющее C»), которое многими операционными системами интерпретируется как указание прервать выполнение программы (таблица 2.6.4).

Для быстрой ориентации в ASCII-кодах запомните содержимое таблицы 2.6.1.

*Таблица 2.6.1*

| **Биты** | | | **Группа символов** |
|---|---|---|---|
| **7** | **6** | **5** | |
| 0 | 0 | 0 | Управляющие символы |
| 0 | 0 | 1 | Цифры и знаки пунктуации |
| 0 | 1 | 0 | Прописные латинские буквы и спецсимволы |
| 0 | 1 | 1 | Строчные латинские буквы и спецсимволы |

Код буквы «Z» – 5Ah (**01011010b**). Чтобы получить код строчной латинской буквы, нужно взять код соответствующей прописной буквы и установить в 5-ом разряде 1. Код буквы «z» – 7Ah (**01111010b**) (таблицы 2.4.3 и 2.6.2).

Коды десятичных цифр 0-9 в любой кодировке (ASCII, EBCDIC, Unicode и т.д.) должны идти в возрастающем порядке, причём коды двух соседних цифр должны отличаться на единицу. Для символов ASCII-цифр значение кода равно самому числу плюс число 30h (таблица 2.6.2). Так как ASCII-цифры следуют в возрастающем порядке, поэтому для сравнения значений применимы арифметические действия непосредственно над кодами чисел.

*Таблица 2.6.2*

**Латинские буквы и цифры в ASCII-кодировке**

| Символы | ASCII код | Биты | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| A | 41h | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| a | 61h | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| B | 42h | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| b | 62h | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| Z | 5Ah | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| z | 7Ah | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 30h | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 31h | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 8 | 38h | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 9 | 39h | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |

*Таблица 2.6.3*

**Основная таблица символов ASCII**

| | **0x** | **1x** | **2x** | **3x** | **4x** | **5x** | **6x** | **7x** |
|---|---|---|---|---|---|---|---|---|
| **x0** | NUL | DLE | | 0 | @ | P | \ | p |
| **x1** | SOH | DC1 | ! | 1 | A | Q | a | q |
| **x2** | STX | DC2 | “ | 2 | B | R | b | r |
| **x3** | ETX | DC3 | # | 3 | C | S | c | s |
| **x4** | EOT | DC4 | $ | 4 | D | T | d | t |
| **x5** | ENQ | NAK | % | 5 | E | U | e | u |
| **x6** | ACK | SYN | & | 6 | F | V | f | v |
| **x7** | BEL | ENTB | / | 7 | G | W | g | w |
| **x8** | BS | CAN | ( | 8 | H | X | h | x |
| **x9** | HT | EM | ) | 9 | I | Y | I | y |
| **xA** | LF | SUB | * | : | J | Z | j | z |
| **xB** | VT | ESC | + | ; | K | [ | k | { |
| **xC** | FF | FS | , | < | L | \ | l | \| |
| **xD** | CR | GS | - | = | M | ] | m | } |
| **xE** | SO | RS | . | > | N | ^ | n | ~ |
| **xF** | SI | US | / | ? | O | _ | o | DEL |

Числа передаются в(из) компьютер(а) в виде последовательности символов, представленных в коде ASCII. Например, число 1234 передается как 31h, 32h, 33h, 34h. Компьютер, принимая число, может запомнить его без модификации, как ASCII-строку; либо обнулить старшие тетрады, что будет соответствовать числу в неупакованном BCD-формате – 01020304h (глава «Двоично-десятичные числа»); либо может удалить старшие тетрады и получить число в упакованном BCD-формате – 1234h; наконец он может преобразовать это число в двоичный формат – 04D2h ($4\times16^2 + 13\times16^1 + 2\times16^0 = 1234$). Выбор того или иного способа преобразования зависит от выполняемой программы.

*Таблица 2.6.3*

**Расширенная таблица символов ASCII (cp866 – русская кодировка в DOS)**

| | 8x | 9x | Ax | Bx | Cx | Dx | Ex | Fx |
|---|---|---|---|---|---|---|---|---|
| **x0** | А | Р | а | ░ | └ | ╨ | р | Ё |
| **x1** | Б | С | б | ▒ | ┴ | ╤ | с | ё |
| **x2** | В | Т | в | ▓ | ┬ | ╥ | т | Є |
| **x3** | Г | У | г | │ | ├ | ╙ | у | є |
| **x4** | Д | Ф | д | ┤ | ─ | ╘ | ф | Ї |
| **x5** | Е | Х | е | ╡ | ┼ | ╒ | х | ї |
| **x6** | Ж | Ц | ж | ╢ | ╞ | ╓ | ц | Ў |
| **x7** | З | Ч | з | ╖ | ╟ | ╫ | ч | ў |
| **x8** | И | Ш | и | ╕ | ╚ | ╪ | ш | ° |
| **x9** | Й | Щ | й | ╣ | ╔ | ┘ | щ | • |
| **xA** | К | Ъ | к | ║ | ╩ | ┌ | ъ | · |
| **xB** | Л | Ы | л | ╗ | ╦ | █ | ы | √ |
| **xC** | М | Ь | м | ╝ | ╠ | ▄ | ь | № |
| **xD** | Н | Э | н | ╜ | ═ | ▌ | э | ¤ |
| **xE** | О | Ю | о | ╛ | ╬ | ▐ | ю | ■ |
| **xF** | П | Я | п | ┐ | ╧ | ▀ | я | |

## 2.6.1. Русские кодировки в DOS и Windows

Символ для вывода на экран или печать описывается с помощью минимальной матрицы 8×16 точек. Данные для каждого символа содержатся в 16 байтах (1 параграф). 256 символов вместе занимают 256×16 = 4096 байта или *кодовую страницу* (CP – *code page*). Адрес матрицы символа соответствует номеру символа в ASCII-кодировке умноженному на 16.

**«Альтернативная кодировка»** – одно из названий кодировки CP866, кодовой страницы, основанной на CP437, где все специфические европейские символы (Ç, Ñ, Ä и так далее) заменены на буквы кириллицы, а псевдографические символы оставлены нетронутыми (таблица 2.6.3).

Существовало несколько вариантов альтернативной кодировки, но все различия касались только символов в области 0F0h - 0FFh.

Окончательным стандартом стала кодировка IBM CP866, поддержка которой была добавлена в MS-DOS версии 6.22. В этой кодировке записываются имена файлов в системе FAT (и короткие имена в VFAT). Сейчас CP866 – стандартная 8-битной русская кодировка Microsoft в среде DOS и OS/2, используется так же в консоли русифицированных систем семейства Windows NT. В Microsoft Windows CP866 заменена стандартной кодировкой CP1251, а в операционных системах Windows NT и следующих за ней (Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008) – кодировкой Unicode.

*Таблица 2.6.5*

**Кодировка code page 1251 (русская кодировка в Windows)**

| | 8x | 9x | Ax | Bx | Cx | Dx | Ex | Fx |
|---|---|---|---|---|---|---|---|---|
| **x0** | Ђ | ђ | | ° | А | Р | а | р |
| **x1** | Ѓ | ‘ | Ў | ± | Б | С | б | с |
| **x2** | ‚ | ’ | ў | І | В | Т | в | т |
| **x3** | ѓ | “ | Ј | і | Г | У | г | у |
| **x4** | „ | ” | ¤ | ґ | Д | Ф | д | ф |
| **x5** | … | • | Ґ | µ | Е | Х | е | х |
| **x6** | † | – | ¦ | ¶ | Ж | Ц | ж | ц |
| **x7** | ‡ | – | § | · | З | Ч | з | ч |
| **x8** | € | □ | Ё | ё | И | Ш | и | ш |
| **x9** | ‰ | ™ | © | № | Й | Щ | й | щ |
| **xA** | Љ | љ | Є | є | К | Ъ | к | ъ |
| **xB** | ‹ | › | « | » | Л | Ы | л | ы |
| **xC** | Њ | њ | ¬ | j | М | Ь | м | ь |
| **xD** | Ќ | ќ | | S | Н | Э | н | э |
| **xE** | Ћ | ћ | ® | s | О | Ю | о | ю |
| **xF** | Џ | џ | Ї | ї | П | Я | п | я |

**ANSI Cyrillic** – одно из названий кодовой страницы Windows-1251 – набор символов и кодировка, являющаяся стандартной 8-битной кодировкой для всех русских версий Microsoft Windows. Создана на базе кодировок, использовавшихся в ранних русификаторах Windows в 1990–1991 гг.

В Windows-1251 присутствуют практически все символы, использующиеся в русской типографике (*типографика* – графическое оформление печатного текста посредством набора и вёрстки с использованием норм и правил, специфических для данного языка) для обычного текста, также содержит символы украинского, белорусского, сербского и болгарского языков.

### 2.6.2. Управляющие символы

Поскольку ASCII изначально предназначался для обмена информацией по телетайпу и телеграфу, в нём, кроме информационных символов (буквы, цифры, знаки препинания), используются *символы-команды* для управления связью. Это набор специальных сигналов, применявшийся в телеграфных и телетайпных средствах обмена сообщениями, дополненный с учётом специфики устройств вычислительной техники.

*Таблица 2.6.6*

**Непечатаемые символы ASCII**

| Код | Имя | Назначение | Ctrl-код |
|---|---|---|---|
| 00 | NUL | Null, *пусто*. Всегда игнорируется. На перфоленте дырка=1, отсутствие дырки=0. Поэтому пустые части перфоленты до начала и после конца сообщения состоят из символов NUL. В C/C++ используется как конец строки. (Строка понимается как последовательность символов.) В некоторых операционных системах NUL – последний символ любого текстового файла. Также используют для временных задержек. | **Ctrl** +@ |
| 01 | SOH | Start of Header, *начало заголовка*. Начинает передачу блока данных или нового файла. Управляющий символ при передаче данных отмечает начало заголовка, если он используется в сообщении наряду с текстом. В заголовке обычно сообщается имя и ячейка с адресом. Этот код первоначально назывался SOM (начало сообщения). | **Ctrl** +A |
| 02 | STX | Start of Text, *начало текста*. Текстом называлась часть сообщения, предназначенная для печати. Адрес, контрольная сумма и т. д. входили или в заголовок, или в часть сообщения после текста. Управляющий символ при передаче данных, используемый в качестве отметки начала текста и конца заголовка (если используется). Этот код первоначально назывался EOA (конец адреса). | **Ctrl** +B |
| 03 | ETX | End of Text, *конец текста*. После символа ETX телетайп прекращал печатать. Может отмечать начало данных при проверке ошибок. Управляющий символ при передаче данных отмечает конец текста. Смотрите символ SOH. Этот код первоначально назывался EOM (конец сообщения) и может использоваться в качестве отметки как таковой на некоторых терминалах. | **Ctrl** +C |
| 04 | EOT | End of Transmission, *конец передачи*. Код остановки, но иногда означает конец файла. Управляющий символ при передаче данных отмечает конец передачи после одного или более сообщений. | **Ctrl** +D |
| 05 | ENQ | Enquire, *прошу подтверждения. Кто там? Запрос*. Запрашивает статусную информацию у отдаленной станции. Управляющий символ при передаче данных обычно используется для идентификации запроса или информации о статусе. В некоторых системах этот код выглядит как WRU (Who aRe yoU? – Кто вы?). | **Ctrl** +E |
| 06 | ACK | Acknowledgment, *подтверждение*. Подтверждает успешный обмен между станциями. Управляющий символ при передаче данных, который служит в качестве обычного ответа «да» на различные вопросы, иногда означает: «Я получил ваше последнее сообщение и готов к вашему следующему сообщению». | **Ctrl** +F |
| 07 | BEL | Bell, *звонок*. Активизирует звонок, гудок или другой звуковой сигнал тревоги на том устройстве, на которое он был послан, чтобы привлечь внимание. В C/C++ обозначается \a. | **Ctrl** +G |

| Код | Имя | Назначение | Ctrl-код |
|---|---|---|---|
| 08 | BS | Backspace, *возврат на один символ*. Управляющий символ перемещает каретку, печатающую головку или курсор назад на один пробел или позицию. Стирает предыдущий символ. В матричных принтерах использовался для создания эффекта жирного или перечеркнутого шрифта ж^Нжи^Нир^Нрн^Нно^Но (**жирно**) и з^Н-а^Н-ч^Н-е^Н-р^Н-к^Н-н^Н-у^Н-т^Н-ь^Н- (~~зачеркнуть~~). | **Ctrl** +H |
| 09 | HT | Horizontal Tabulation, *горизонтальная табуляция.* Управляющий символ, который устанавливает каретку, печатающее колесико или курсор на следующем заранее определенном месте той же строки. Пользователь обычно решает, как осуществлять разметку горизонтальной таблицы. В C/C++ обозначается \t. | **Ctrl** +I |
| 0A | LF | Line Feed, *перевод строки*. Управляющий символ перемещает каретку, печатающую головку или курсор вниз на одну строку. Сейчас в конце каждой строчки текстового файла ставится либо этот символ, либо CR, либо и тот и другой (CR, затем LF), в зависимости от операционной системы. В C/C++ обозначается \n и при выводе текста приводит к переводу строки. | **Ctrl** +J |
| 0B | VT | Vertical Tabulation, *вертикальная табуляция.* Управляющий символ, который подводит каретку, печатающую головку или курсор к следующему заранее определенному ограничителю (обычно строке). | **Ctrl** +K |
| 0C | FF | Form Feed, *новая страница*. Перевод формата. | **Ctrl** +L |
| 0D | CR | Carriage Return, *возврат каретки*. Управляющий символ перемещает каретку, печатающую головку или курсор на экране терминала к началу данной строки. В C/C++ этот символ, обозначается \r, его используют для возврата в начало строчки без перевода строки. В некоторых операционных системах этот символ ставится в конце каждой строчки текстового файла перед LF. | **Ctrl** +M |
| 0E | SO | Shift Off, *сдвиг выключен*. Изменить цвет ленты (использовался для двуцветных лент печатной машинки – терминала ЭВМ; цвет менялся обычно с черного на красный). Позднее стало обозначать начало использования национальной кодировки, переключение набора символов, переключение на дополнительный регистр. Управляющий символ указывал, что значения следующих двоичных образов выходят за рамки стандартного набора символов кода ASCII до тех пор, пока не будет введен символ переключения на стандартный регистр. | **Ctrl** +N |
| 0F | SI | Shift In, *сдвиг включен*. Переключение набора символов, переключение на стандартный регистр. Управляющий символ, используемый после кода SO для указания того, что коды возвращаются к обычному значению в коде ASCII. | **Ctrl** +O |

| Код | Имя | Назначение | Ctrl-код |
|---|---|---|---|
| 10 | DLE | Data Link Escape, *оставить канал данных*. Модифицирует значение следующих символов (аналогично Esc). Управляющий символ при передаче данных, который использует специальный тип управляющей последовательности специально для управления каналом связи и средствами передачи. | **Ctrl** +P |
| 11 | DC1/ XON | Device Control 1, *первый символ управления устройством* – включить устройство чтения перфоленты. «XON» – дословно означает «включить передатчик». Используется как сигнал удаленной станции на передачу. | **Ctrl** +Q |
| 12 | DC2/ TAPE | Device Control 2, *второй символ управления устройством* – включить перфоратор. Сигнал переключения общего назначения. | **Ctrl** +R |
| 13 | DC3/ XOFF | Device Control 3, *третий символ управления устройством* – выключить устройство чтения перфоленты. Используется как сигнал XOFF удаленной станции на прекращение передачи. | **Ctrl** +S |
| 14 | DC4/ NO TAPE | Device Control 4, *четвертый символ управления устройством* – выключить перфоратор. Сигнал переключения общего назначения. | **Ctrl** +T |
| 15 | NAK | Negative Acknowledgment, *не подтверждаю*. Отрицательное подтверждение. Передача неуспешна. Управляющий символ при передаче данных указывает «нет» в ответе на различные вопросы. Иногда описывается как: «Я получил ваше последнее сообщение, но в нем есть ошибки, и я жду повторного сообщения». | **Ctrl** +U |
| 16 | SYN | Synchronization, *синхронизация*. Используется между блоками для синхронной связи. Управляющий символ при передаче данных, используемый некоторыми быстродействующими системами передачи данных, которые используют синхронизированные часы на концах приемника и передатчика. В периоды простоя, когда отсутствие двоичных образов не позволяет часам на приемном конце отслеживать часы передатчика, приемник может нарушать синхронизацию. Каждая передача, следующая за период простоя, поэтому заменяются тремя или четырьмя символами синхронизации SYN. Код SYN имеет двоичный образ, который позволяет приемнику не только запирать (блокировать) часы передатчика, но также определять точки начала и конца каждого символа. Символы синхронизации SYN могут, кроме того, использоваться для заполнения коротких периодов простоя с тем, чтобы поддерживать синхронизацию, отсюда и данное название. | **Ctrl** +V |
| 17 | ENTB | End of Text Block, *конец текстового блока*. Иногда текст по техническим причинам разбивался на блоки. Конец блока передачи. Вариант ETX. | **Ctrl** +W |

| Код | Имя | Назначение | Ctrl-код |
|---|---|---|---|
| 18 | CAN | Cancel, *отмена* (того, что было передано ранее). Обычно сигнализирует об ошибке передачи. Управляющий символ общего назначения, указывающий что информация предыдущей передачи не должна учитываться, причем объем этой информации определяется пользователем. | **Ctrl** +X |
| 19 | EM | End of Medium, *конец носителя*. Конец блока передачи сигнализирует о физическом конце источника данных – кончилась перфолента и т.д. | **Ctrl** +Y |
| 1A | SUB | Substitute, *подстановка*. Заменяет ошибочные символы или сигнализирует, что следующий символ другого цвета или из дополнительного набора символов. В системах DOS и Windows этот символ используется как конец файла при вводе с клавиатуры. | **Ctrl** +Z |
| 1B | ESC | Escape – отмечает, что последующие символы – управляющая последовательность. Отмечает начало управляющей последовательности, состоящей из серии кодов, которые в качестве группы имеют особое назначение, обычно функции управления. В некоторых терминалах символ ESC назывался ALT MODE, соответствовавшей современной клавише Alt на клавиатуре. | **Ctrl** +[ |
| 1C | FS | File Separator, *разделитель файлов*. Отмечает логическую границу между файлами. | **Ctrl** +\ |
| 1D | GS | Group Separator, *разделитель групп*. Отмечает логическую границу между группами данных. | **Ctrl** +] |
| 1E | RS | Record Separator, *разделитель записей*. Отмечает логическую границу между записями данных. | **Ctrl** + ^ |
| 1F | US | Unit Separator, *разделитель элементов*. Отмечает логическую границу между объектами данных. То есть поддерживалось 4 уровня структуризации данных: сообщение могло состоять из файлов, файлы из групп, группы из записей, записи из элементов. | **Ctrl** +_ |
| 7F | DEL | Delete, *стереть последний символ*. Символом DEL, состоящим в двоичном коде из всех единиц, можно было забить любой символ. Устройства и программы игнорировали символ DEL так же, как NUL. На перфоленте удаление символа происходило забиванием его кода дырочками (обозначавшими логические единицы). | **Ctrl** +? |

Любой код ASCII, кроме 0, может быть введен с клавиатуры путем нажатия клавиши «Alt», набора кода ASCII на дополнительной клавиатуре и затем отпускания клавиши «Alt» (режим «Num Lock» должен быть включен).

Комбинации клавиш «Ctrl» с буквами латинского алфавита генерируют однобайтовые коды ASCII. Код большой латинской буквы должен быть на 64 больше, чем представляемого таким образом управляющего символа. Например, «перевод строки» LF = **Ctrl** + «J» (4Ah – 40h = 0Ah).

Под это правило не попадает код DEL = **Ctrl** + «?» (3Fh + 40h = 7Fh).

Имеется 4 кода ASCII, которые могут быть получены еще одним способом:

| Код | Имя | Ctrl-код | Клавиша |
|---|---|---|---|
| 08 | BS | **Ctrl** +H | Backspace |
| 09 | HT | **Ctrl** +I | Tab |
| 0D | CR | **Ctrl** +M | Enter |
| 1B | ESC | **Ctrl** +[ | Esc |

Символы, соответствующие управляющим кодам ASCII, не выводятся на экран при использовании функций ввода с клавиатуры. Они могут быть выведены либо с помощью функции 10h прерывания BIOS (10h), либо прямым выводом в видеопамять.

## 2.7. Строка

Непрерывный набор смежных байтов, слов или двойных слов максимальной длиной до 4 Гбайт.

### Контрольные вопросы и упражнения

1. Сколько бит находится в байте, слове, двойном и учетверенном слове?
2. Составьте таблицу целых однобайтовых чисел (от 1 до 20), представленных в различных системах счисления (десятичной, двоичной, восьмеричной, шестнадцатеричной).
3. Составьте таблицу максимальных и минимальных целых чисел (со знаком и без знака), помещающихся в один байт и представленных в следующих системах счисления:

   а) десятичной; б) двоичной; в) восьмеричной; г) шестнадцатеричной.

4. Подсчитайте диапазон значений для двойного слова без знака.
5. Преобразуйте из одной системы счисления в другую по схеме: десятичная→шестнадцатеричная→двоичная→восьмеричная→ десятичная заданные числа:

   1) 270; 5) 525; 9) 321; 13) 444;
   2) 303; 6) 451; 10) 501; 14) 218;
   3) 472; 7) 268; 11) 495; 15) 381;
   4) 539; 8) 399; 12) 217; 16) 505.

6. Почему максимальное положительное число в 8-разрядном дополнительном коде равно 127, а максимальное отрицательное –128?

7. Что такое основание системы счисления?
8. Почему для отображения данных в основном используют шестнадцатеричную систему счисления?
9. Как представлены в памяти числа без знака и со знаком?
10. Что такое дополнение до двух?
11. Преобразовать следующие десятичные числа в двоичные:
    а) 19, б) 79, в) 463, г) 1209, д) 11355.
12. Преобразовать следующие двоичные числа в десятичные:
    а) 10110b, б) 1011011b, в) 10010010b, г) 101100011110110b.
13. Преобразовать следующие числа в шестнадцатеричные:
    а) 35, б) 298, в) 852, г) 162023, д) 10110b, е) 10010010b, г) 101100011110110b.
14. Найти двоичные и десятичные эквиваленты следующих шестнадцатеричных чисел:
    а) 0D5h, б) 9A26h, в) 7BF52Ah, г) 2A01BF57h.
15. На вашей клавиатуре сломались кнопки Enter, Tab и Backspace. Как вы выйдете из данного положения?
16. Сложите числа 11110001b и 01001010b, воспринимая их:
    а) как числа без знака; б) как числа со знаком. Выполните проверку, осуществив те же действия в десятичной системе счисления.
17. Выполните действия с данными числами, предварительно переведя их в двоичную систему счисления:

| | | | |
|---|---|---|---|
| 255+127 | 127+22 | 250+6 | 127+128 |
| 32767–250 | 8–256 | 0–32768 | 256–8 |
| –128+127 | –1+128 | –128+256 | –32768+220 |
| –128–25 | –32768–0 | –127–2 | –127–22 |
| 26×5 | 7×8 | 10×20 | 5×6 |
| 6×(–8) | –6×21 | –6×6 | 25×(–5) |
| 130/5 | 56/8 | 200/10 | 30/5 |
| –48/8 | 126/(–6) | 36/(–6) | –125/5 |

    Проверьте результат, осуществив те же действия в десятичной системе счисления.

18. Пусть A=00110010b, B=01001010b, C=11101001b, D=10111010b.
    Выполнить следующие операции в дополнительном коде:

а) A+B, б) A+C, в) C+B, г) C+D, д) A–B, е) C–A, ж) D–C, з) A+D–C.

Выполните проверку, осуществив те же действия в десятичной системе счисления.

19. Найти чему равно
    а) 9h+1h б) 10h–1h
20. Пусть A=0561h, B=0352h, C=9B6Fh, D=917Dh. Выполнить следующие операции в дополнительном коде:
    а) A+B, б) A+C, в) C+D, г) A–B, д) C–D, е) B–C.
21. Сложите следующие двоичные числа:
    а) 00010101b+00001101b, б) 00111110b+00101001b,
    в) 00011111b+00000001b.
22. Найдите двоичные дополнения для следующих двоичных чисел:
    а) 00010011b, б) 00111100b, в) 00111001b.
23. Найдите положительные значения для следующих отрицательных двоичных чисел:
    а) 11001000b, б) 10111001b, в) 10000000b.
24. Найдите двоичные дополнения для следующих шестнадцатеричных чисел:
    а) 3251h, б) 12907h.
25. Какие из следующих шестнадцатеричных чисел отрицательные?
    а) 0F100h, б) 0155h, в) 0A011h, г) 0005h, д) 71FFh?
26. Пусть *n* – некоторое десятичное 14-значное целое число. Можно ли поместить значение *n* в 48-битное машинное слово, состоящее из 47 битов под значение плюс 1 бит под знак?
27. Даны девять чисел: 254, 128, 248, 240, 192, 224, 252, 0, 255. Как расположить эти числа на плоскости, чтобы образовались два прямоугольных треугольника?

    О т в е т : если числа рассортировать по возрастанию, разместить одно под другим, а затем перевести их в двоичный вид:

0 =00000000
128=10000000
192=11000000
224=11100000
240=11110000
248=11111000
252=11111100
254=11111110

255=11111111

то, как видно, образуются два прямоугольных треугольника. Разумеется числа можно расположить и в обратном порядке.

28. Нарисуйте «елочку» числами в двоичном коде.
29. Заполните таблицу

| Двоичное число | Шестнадцатеричное число | Десятичное число |
|---|---|---|
| 100b | | |
| 10101101b | | |
| 1101110101b | | |
| 11111011110b | | |
| 10000000001b | | |
| | 8EFh | |
| | 10h | |
| | 0A52Eh | |
| | 70Ch | |
| | 6BD3h | |
| | | 100 |
| | | 527 |
| | | 4128 |
| | | 1194 |
| | | 59020 |

ГЛАВА 3

# АРХИТЕКТУРА 32/64-РАЗРЯДНОГО МИКРОПРОЦЕССОРА СЕМЕЙСТВА 80X86

## 3.1. Память

Программы, выполняемые процессором, находятся не в воздухе и даже не в самом процессоре, а в оперативной памяти компьютера. Процессор забирает из памяти очередную команду, выполняет ее, потом переходит к следующей команде, снова выполняет ее – и так до конца программы. Команды процессора могут не только менять его собственное содержимое (содержимое регистров процессора), но и записывать числа в память компьютера. Память компьютера делится на ячейки размером 8 разрядов. Ячейки такого размера называют *байтом*. Разряды байта нумеруются справа налево от 0 до 7 (рис. 3.1). При этом правые разряды с меньшими номерами называются *младшими разрядами*, а левые разряды – *старшими*. В каждый разряд можно записать величину 0 или 1, такую величину называют *битом*. Самому первому байту присвоен нулевой номер. Номер последнего байта определяется объемом оперативной памяти, которой располагает компьютер. Порядковый номер байта называется *адресом*.

Объем оперативной памяти измеряется в байтах. Единичные и нулевые биты на самом деле – математическая абстракция. Для процессора реальны только напряжения на его внешних контактах. Каждый контакт соответствует одному биту и процессору нужно различать только две градации напряжения: «высокий» уровень напряжения и «низкий» уровень напряжения. Одному уровню напряжения соответствует «единица», другому – «ноль». Поэтому адрес для процессора – это последовательность напряжений на специальных контактах, называемых *шиной адреса*. Объем физически адресуемой микропроцессором памяти однозначно определяется разрядностью внешней шины адреса как $2^N$, где N – количество адресных линий. Поэтому объем памяти компьютера определяется в числах, кратных степеням двойки:

1kB (килобайт, греч. *chilioi* тысяча) = $2^{10} \approx 10^3$ байт;

1MB (мегабайт, греч. *megas* большой) = $2^{20} \approx (10^3)^2$ байт;

1GB (гигабайт, греч. *gigas* гигантский) = $2^{30} \approx (10^3)^3$ байт;

1TB (терабайт, греч. *teratos* чудовище) = $2^{40} \approx (10^3)^4$ байт;

1PB (петабайт, греч. *pente* пять, тысяча в 5-ой степени) = $2^{50} \approx (10^3)^5$ байт;

1EB (эксабайт, греч. *hex* шесть, тысяча в 6-ой степени) = $2^{60} \approx (10^3)^6$ байт;

1ZB (зеттабайт, тысяча в 7-ой степени) = $2^{70}$ байт $\approx (10^3)^7$ байт;

1JB (йотабайт, тысяча в 8-ой степени) = $2^{80}$ байт $\approx (10^3)^8$ байт.

*Таблица 3.1*

| Единица измерения | $2^N$ | В байтах | В килобайтах | В мега-байтах | В гига-байтах |
|---|---|---|---|---|---|
| килобайт | $2^{10}$ | 1024 | 1 | – | – |
| мегабайт | $2^{20}$ | 1048576 | 1.024 | 1 | – |
| гигабайт | $2^{30}$ | 1073741824 | 1.048.576 | 1.024 | 1 |
| терабайт | $2^{40}$ | 1099511627776 | 1.073.741.824 | $1.024^2$ | 1.024 |
| петабайт | $2^{50}$ | 1125899906842624 | $1.024^4$ | $1.024^3$ | $1.024^2$ |
| эксабайт | $2^{60}$ | 1152921504606846976 | $1.024^5$ | $1.024^4$ | $1.024^3$ |
| зеттабайт | $2^{70}$ | $1024^7$ | $1.024^6$ | $1.024^5$ | $1.024^4$ |
| йотабайт | $2^{80}$ | $1024^8$ | $1.024^7$ | $1.024^6$ | $1.024^5$ |

*Старая шутка*: начинающий программист считает, что в килобайте 1000 байтов, а законченный программист – что в килограмме 1024 грамма.

Адресом машинного слова является адрес его младшего байта. В памяти младший байт всегда хранится по меньшему адресу. Адрес старшего байта может быть использован для доступа к старшей половине слова.

Адресом двойного слова является адрес его младшего слова. Адрес старшего слова может быть использован для доступа к старшей половине двойного слова.

Адресом учетверенного слова является адрес его младшего двойного слова. Адрес старшего двойного слова может быть использован для доступа к старшей половине учетверенного слова.

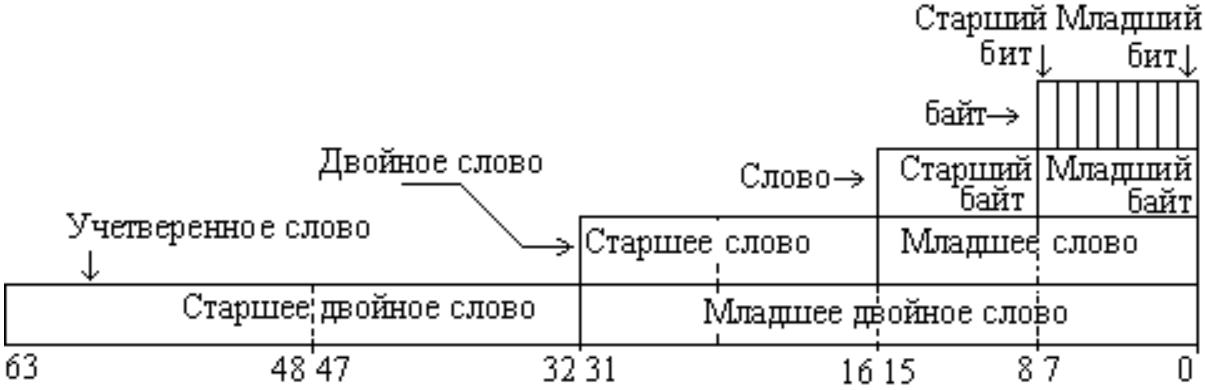



***Рис. 3.1***

Кроме контактов, на которых появляется адрес, в процессоре есть еще контакты, называемые *шиной данных*, где появляется прочитанное из памяти число. Код программы и ее данные находятся в одном адресном пространстве. Размещение данных и кода программ в памяти осуществляется самой системой в процессе работы и в различные моменты может быть различным; то, что было в течение некоторого времени пространством программы (командами), в другое время может стать массивом

данных. Сама по себе память в этом случае имеет совершенно общий характер. При использовании микропроцессорных устройств для решения конкретных практических задач, назначение памяти закладывается в конструкцию устройства. Блоки энергонезависимой памяти (*ПЗУ* – постоянное запоминающее устройство) «только для чтения» используются для хранения программы, а энергозависимая память (*ОЗУ* – оперативное запоминающее устройство) для считывания/записи – для временного хранения данных, организации стеков и как рабочее пространство программы. Использование энергонезависимой памяти для хранения программ является общим правилом при конструировании специального оборудования, поскольку позволяет избежать ввода программы в такие устройства при каждом их включении.

## 3.2. Внутренняя архитектура 32/64-разрядного микропроцессора семейства 80x86

Внутренняя архитектура 32/64-разрядного микропроцессора семейства 80x86 представлена на рис. 3.2.

В базовый состав микропроцессора входит арифметико-логическое устройство (АЛУ), которое выполняет такие операции, как сложение, дополнение, сравнение, пересылка, сдвиг и так далее над данными, которые хранятся в регистрах или во внутренней памяти компьютера. Для увеличения скорости выполнения программ микропроцессор Pentium содержит два арифметико-логических устройства, благодаря чему две команды могут быть выполнены за один такт синхронизации.

Регистры – это сверхбыстродействующая память, которая физически находится в микропроцессоре. Регистры используются для вычислений и связи микропроцессора с внешним миром. За исключением счетчика команд (регистр RIP/EIP) и 20-байтной очереди команд, регистры управления и рабочие регистры разделены на три группы в соответствии с выполняемыми им функциями. Имеется группа регистров данных (регистры общего назначения, РОН), представляющая собой, по существу, набор арифметических регистров; указательная группа (регистры RSP/ESP, RSI/ESI, RDI/EDI, RBP/EBP), содержащая базовые и индексные адреса; сегментная группа, в состав которой входят специальные базовые регистры (регистры CS, DS, SS, ES, FS и GS).

Устройство управления и синхронизации обеспечивает связь микропроцессора x86 с памятью и устройствами ввода/вывода.

Для управления памятью микропроцессор использует расширенное устройство управления памятью (Memory Management Unit, MMU).

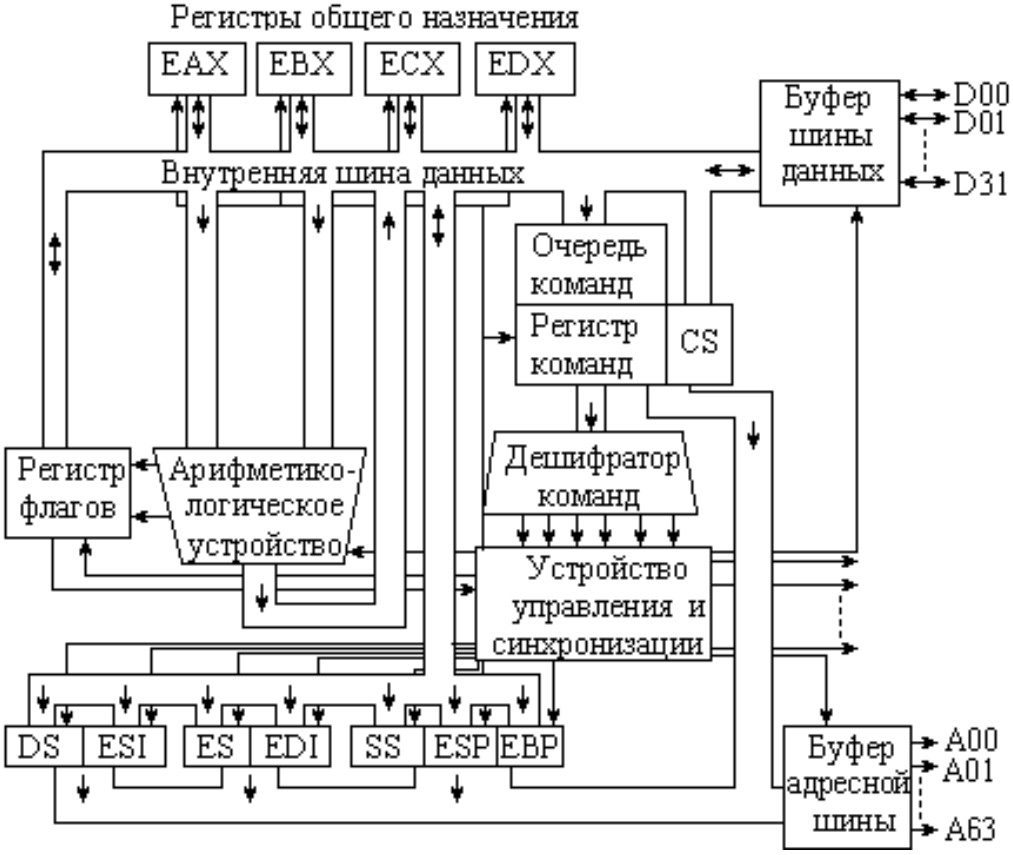



***Рис. 3.2. Внутренняя архитектура микропроцессора x86***

### 3.2.1. Цикл выполнения команды

Под *циклом выполнения команды* подразумевается последовательность действий, совершаемых микропроцессором при выполнении одной машинной команды. При выполнении команды, в которой используется операнд, расположенный в памяти, микропроцессор должен сначала определить адрес операнда, поместить его на шину адреса, дождаться, пока значение операнда появится на шине данных.

Перед выполнением программа должна быть загружена в память компьютера. Упрощенная схема цикла выполнения команды показана на рисунке 3.3. Процессор берёт из памяти машинную команду и увеличивает текущий адрес так, чтобы он указывал на следующую команду. Адрес следующей по порядку выполняемой команды помещается в регистр EIP (Счетчик команд). Очередь команд – область сверхоперативной памяти внутри микропроцессора, в которую помещена одна или несколько команд непосредственно перед выполнением. При выполнении каждой команды микропроцессор выполняет следующие операции: *выборку*, *декодирование* и *выполнение*. Если в команде используется операнд, расположенный в памяти, микропроцессору придется дополнительно выполнить *выборку операнда из памяти* и *запись результата в память*.

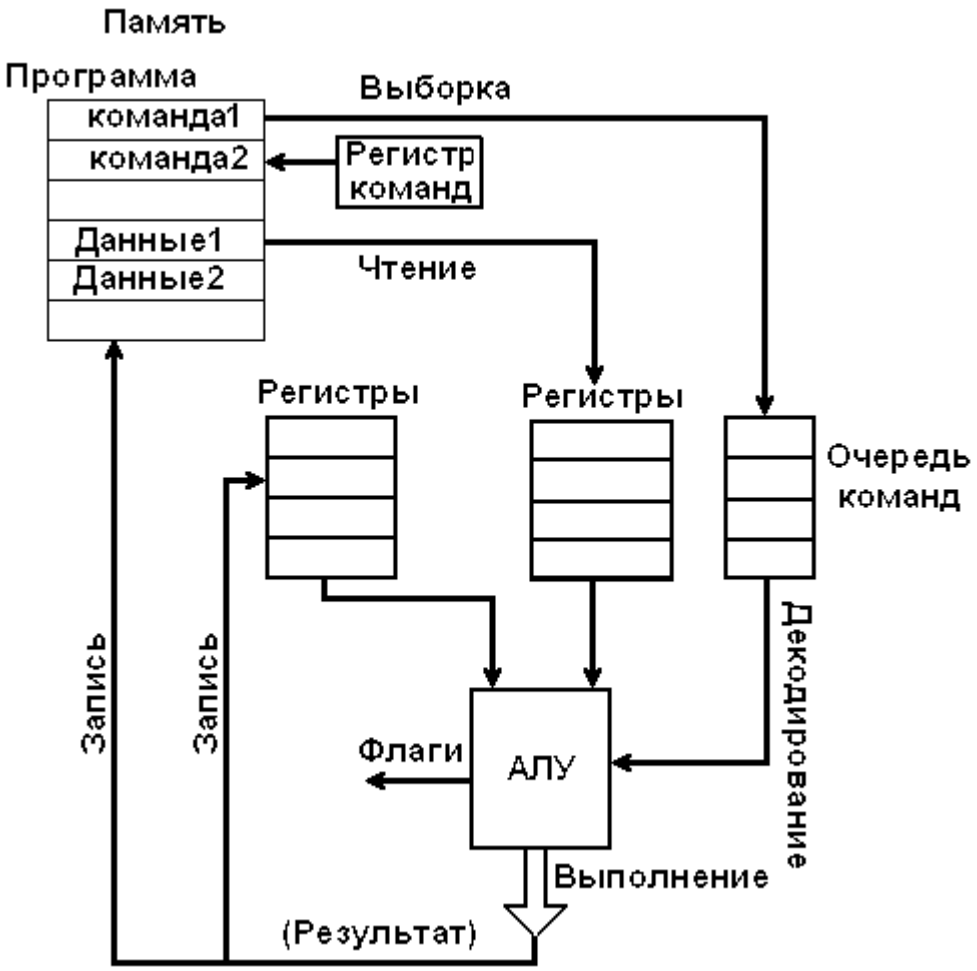



***Рис. 3.3. Упрощенная схема цикла выполнения команды***

*Выборка команды*. Блок управления извлекает команду из памяти, копирует ее во внутреннюю память микропроцессора и увеличивает значение счетчика команд на длину этой команды.

*Декодирование команды*. Блок управления определяет тип выполняемой команды, пересылает указанные в ней операнды в АЛУ и генерирует сигналы управления АЛУ, соответствующие типу выполняемой операции.

*Выборка операндов*. Если в команде используется операнд, расположенный в памяти, блок управления инициализирует операцию по выборке его из памяти.

*Выполнение команды*. АЛУ выполняет указанную в команде операцию, сохраняет полученный результат в заданном месте и обновляет состояние регистра флагов. По значению флагов программа может судить о результате выполнения команды.

*Запись результата в память*. Если результат выполнения команды должен быть сохранен в памяти, блок управления инициирует операцию сохранения данных в памяти.

## 3.3. Регистры микропроцессора

Универсальные регистры являются составной частью процессора. Они используются для временного хранения информации. Интенсивное использование регистров микропроцессором при работе с программой определяется тем, что скорость доступа к ним намного больше, чем к ячейкам памяти. 32-/64-битные процессоры имеют набор регистров для хранения данных общего назначения; набор сегментных регистров; набор из 8 80-битных регистров для работы с числами с плавающей точкой (ST0-ST7); набор из 8 64-битных регистров целочисленного MMX-расширения (MMX0-MMX7, имеющих общее пространство с регистрами ST0-ST7); набор из 16 128-битных регистров SSE для работы с числами с плавающей точкой (XMM0–XMM15); программный стек – специальная информационная структура, работа с которой предусмотрена на уровне машинных команд. Помимо основных регистров из реального режима доступны также регистры управления памятью (GDTR, IDTR, TR, LDTR) регистры управления (CR0-CR4), отладочные регистры (DR0-DR7) и машинно-специфичные регистры.

### 3.3.1. Регистры общего назначения

Различают 8 целочисленных 32-х разрядных регистров общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) их 64-разрядные расширения (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP) и 8 целочисленных 64-разрядных регистров общего назначения (R8-R15), которые могут хранить следующие типы данных:

- операнды для логических и арифметических операций;
- операнды для расчета адресов;
- указатели на ячейки памяти.

Хотя для хранения операндов, результатов операций и указателей Вы можете использовать любой из вышеперечисленных регистров, будьте осторожны с регистром ESP/RSP. В нем хранится указатель вершины стека, и некорректное изменение этого значения приведет к неправильной работе программы и ее аварийному завершению.

Многие команды используют конкретные регистры для хранения своих операндов. Например, команды обработки текстовых строк используют содержимое регистров ECX/RCX, ESI/RSI и EDI/RDI в качестве операндов.

Основные случаи использования регистров общего назначения:

- EAX/RAX – используется для хранения операндов и результатов операций;
- EBX/RBX – как указатель на данные в сегменте DS;
- ECX/RCX – как счетчик для строковых операций и циклов;

- EDX/RDX – указатель для ввода/вывода, по умолчанию регистры EAX/RAX и EDX/RDX используются для умножения и деления;
- ESI/RSI – указатель на данные в сегменте DS, а также как указатель на источник в командах работы со строками;
- EDI/RDI – указатель на данные в сегменте ES, а также как указатель на приемник в командах работы со строками;
- ESP/RSP – указатель вершины стека в сегменте SS;
- EBP/RBP – указатель на некоторые данные в стеке.

К регистрам, оканчивающимся на X, можно обращаться и как целиком к 64-битным (RAX, RBX, RCX, RDX), и к их младшей 32-битной части (EAX, EBX, ECX, EDX), и к младшим 16-ти битам (AX, BX, CX, DX), которые в свою очередь можно разделить на старший/младший байт (AH/AL, BH/BL, CH/CL, DH/DL) и работать с ними, как с регистрами длиной 8 бит. Регистры-указатели RSP/ESP (указатель вершины стека) и RBP/EBP (базовый регистр), а также индексные регистры RSI/ESI (индекс источника) и RDI/EDI (индекс приемника) допускают либо 64-/32-битное обращение, либо обращение только к младшим 16 битам (SP, BP, SI и DI), либо к младшим 8 битам (SPL, BPL, SIL, DIL). Для обращения к младшим 8/16/32-битам регистров R8-R15 используют суффиксы b/w/d. Например, R9b – младший байт 64-разрядного регистра R9 (по аналогии с AL), R9w – младшее слово регистра R9 (то же, что AX в EAX). Для обращения к старшей части регистров РОН используют сдвиги или математические операции.

### 3.3.2. Регистры сегментов

Регистры сегментов (CS, DS, SS, ES, FS и GS) хранят 16-битные базовые адреса сегментов, определяющие сегменты памяти текущей адресации.

В защищенном режиме каждый сегмент может иметь размеры от одного байта до целого линейного и физического пространства машины до 4 Гб в режиме реальной адресации, максимальный размер сегмента ограничен на 64 Кб.

Шесть сегментов, адресуемых в любой данный момент, определяются содержимым регистров CS, SS, DS, ES, FS и GS:

- значение в CS (*Code Segment Register*) указывает на сегмент, содержащий команды программы;
- содержимое DS (*Data Segment Register*) указывает на сегмент, содержащий обрабатываемые программой данные;
- содержимое SS (*Stack Segment Register*) сегмент, содержащий стек программы;
- значения в ES, GS указывают на дополнительные сегменты данных;
- регистр FS содержит блок данных потока (thread information block TIB). Этот блок описывает выполняющийся в данный момент поток.

### 3.3.3. Обрабатывающая часть микропроцессора

Основой микропроцессора является арифметико-логическое устройство (АЛУ), предназначенное для обработки информации. АЛУ содержит две входные (A и B) и одну выходную (F) 64-/32-разрядные шины. Информация на и входных шинах обрабатывается в АЛУ в соответствии с набором управляющих сигналов на управляющих шинах АЛУ. Результат обработки появляется на выходной шине.

Другим важным узлом микропроцессора является набор регистров общего назначения (РОН). В РОН хранятся информационные слова, подлежащие обработке в АЛУ, результаты обработки информации в АЛУ и управляющие слова. Обращение к РОН – адресное.

Так как в один и тот же момент можно обращаться к 8-битным, либо 16-битным, либо к 32- или 64-битным регистрам, поэтому адреса регистров с разными размерами совпадают. Для формирования оптокода используются следующие номера регистров.

*Таблица 3.3.1*

| 8-битный операнд | 16-битный операнд | 32-битный операнд | 64-битный операнд | Адреса регистров в двоичном формате |
|---|---|---|---|---|
| AL | AX | EAX | RAX | 000b |
| CL | CX | ECX | RCX | 001b |
| DL | DX | EDX | RDX | 010b |
| BL | BX | EBX | RBX | 011b |
| AH/SPL | SP | ESP | RSP | 100b |
| CH/BPL | BP | EBP | RBP | 101b |
| DH/DIL | SI | ESI | RSI | 110b |
| BH/SIL | DI | EDI | RDI | 111b |

Эти регистры допускают считывание и запись информации, поэтому содержат входную и выходную шины данных, адресную шину и управляющие входы, информация на котором задает режим работы: запись, хранение или чтение информации.

На рис. 3.3.1 типовая схема обрабатывающей части микропроцессора. Содержимое любого РОН может быть передано на буферный регистр (БР) и на регистр сдвига ($Р_{сдв}$). АЛУ выполняет логические и арифметические операции над содержимым обоих регистров; результат может быть записан в любой из РОН. При подаче соответствующих управляющих сигналов в этой системе, возможны:

– передача данных из одного РОН в другой путем пересылки выбранного из первого РОН слова транзитом через БР и АЛУ во второй РОН;

– увеличение или уменьшение содержимого любого РОН путем изменения в АЛУ выбранного из РОН значения на единицу и засылки полученного результата в тот же регистр;

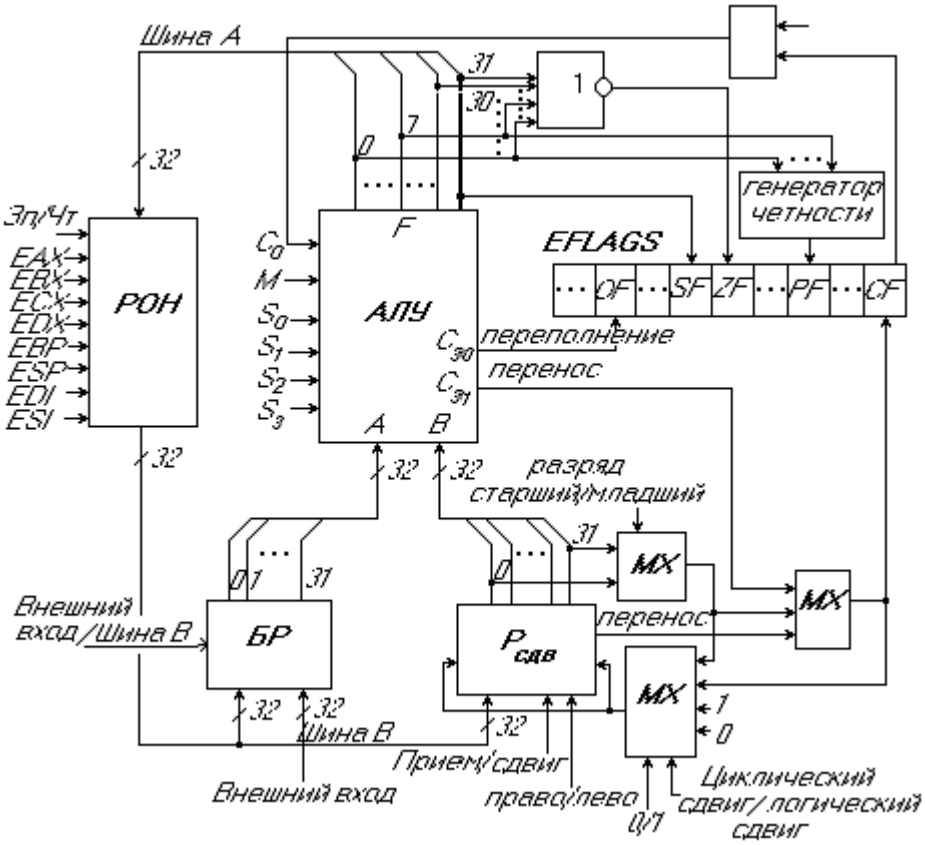



***Рис. 3.3.1***

– сдвиг содержимого любого РОН путем передачи выбранного из РОН значения в $Р_{сдв}$, сдвига этого числа и записи через АЛУ в тот же регистр.

Образующийся при выполнении арифметических операций сигнал переноса хранится в триггере CF, там же сохраняется значение вытесняемого при сдвиге разряда из $Р_{сдв}$. Выходной код АЛУ может быть проанализирован на «все нули» (ZF) и на значение старшего знакового разряда (SF).

Общее количество внутренних регистров микропроцессора зависит от его модели. Первоначально (до появления суперскалярного микропроцессора) регистры были не виртуальными, на один программный регистр приходилось по одному внутреннему регистру, затем – по 2 штуки внутренних регистров на один программный и так далее. Количество внутренних регистров (RF) больше, чем программных. Регистры RF не привязаны к программным регистрам, то есть говорить «*регистров EAX – 8 штук*» или «*по восемь на каждый*» – не правильно. Регистры RF привязаны к операциям (точнее к микрооперациям – *мопам*) – каждому мопу назначается новый регистр RF для записи результата. Поэтому и количество регистров зависит от того, сколько мопов могут одновременно находиться в конвейере процессора – чем длиннее конвейер, тем больше должно быть RF и ROB (reorder buffer, в котором временно хранятся

мопы). Поскольку современные процессоры суперскалярные и рассчитаны на обработку в среднем до трех мопов за такт (Core 2 Duo до 4), то количество RF и ROB должно быть не меньше утроенной задержки (в тактах) прохождения мопа по конвейеру с выхода декодера (или Т-кэша) до выхода в отставку (удаления из ROB). Например, в Pentium Pro-Pentium III количество RF и ROB – 40, а в Pentium 4E конвейер в 3 раза длиннее и соответственно RF и ROB – 128.

### 3.3.4. Сегментация памяти

В 1976 году фирма Intel закончила разработку 16-разрядного микропроцессора i8086. Он имел достаточно большую разрядность регистров (16 бит) и системной шины адреса (20 бит), за счет чего микропроцессор мог адресовать до 1 Мбайта оперативной памяти. Использование сегментных регистров в то время называлось на компьютерном жаргоне словом *kludge* (приспособление для временного устранения проблемы). Проблема заключалась в адресации более 64 Кбайт памяти – предела, который устанавливается использованием 16-битных регистров, так как $2^{16}$=65535 – это наибольшее число, которое может содержать такой регистр. Специалисты фирмы Intel для решения этой проблемы использовали сегменты и регистры сегментов и тем самым усложнили процесс адресации в микропроцессорах x86. Хотя адрес формируется из двух 16-разрядных регистров, микропроцессор i8086 (а после i80186, i80286) не формировал для адреса 32-разрядное число, (он показался инженерам Intel чрезмерно большим), а обходился 20-разрядным адресом. Микропроцессор разбивает память на перекрывающиеся *сегменты*. *Сегмент* – это 64-Кбайтный участок памяти. Каждый новый сегмент начинается через каждые 16 байт. Первый сегмент (сегмент #0) начинается в памяти с ячейки #0(=0000.0000b); второй (сегмент #1) начинается с ячейки #16 (10h=0001.0000b); третий – с ячейки #32 (20h=0010.0000b) и так далее. То есть начальный адрес сегмента памяти всегда кратен 16 и последние 4 бита у этого адреса нулевые. В сегментных регистрах хранят только первые 16 битов начального адреса сегмента памяти. Для микропроцессора i8086 полный 20-разрядный (*абсолютный* или *физический*) адрес получался сложением содержимого сегментного регистра, умноженное на 16 (начального адреса сегмента) и содержимого какого-нибудь 16-разрядного регистра общего назначения, указывающего на смещение внутри сегмента (*эффективный адрес* или *смещение*).

$$\text{Физический адрес} = \text{сегмент} \times 16 + \text{смещение} \qquad (2)$$

Например, адресная пара регистр ES=1234h и регистр DI=0053h задают следующий абсолютный адрес:

$$ES:DI=10h\times1234h+0053h=12340h+53h=12393h.$$

Обратите внимание, что при других значениях в ES=1000h и DI=2393h мы получим тот же абсолютный адрес:

ES:DI=10h×1000h+2393h=10000h+2393h=12393h.

При сложении максимальных значений сегмента и смещения формула 2 даст адрес 0FFFF0h+0FFFFh=10FFEFh, но из-за 20-разрядного ограничения на шину памяти эта комбинация указывает на адрес 0FFEFh. Таким образом, у нас получается совмещение адресов – адреса от 100000h до 10FFEFh соответствовали адресам от 0 до 0FFEFh. Начиная с i80286 шина адреса увеличена до 24 разрядов, i80386 до 32, а с Pentium Pro до 36 разрядов. Меняются и способы адресации к памяти. Микропроцессор x86 позволяет 24 способа адресации.

### 3.3.5. Защищенный режим и виртуальная память

Для того чтобы процессы не мешали друг другу (по недосмотру или умышленно), требуются меры принудительной защиты критических ресурсов. Современные операционные системы используют *защищенный режим* процессора, в котором эти меры реализуются на аппаратном уровне. Поскольку программа может взаимодействовать с подсистемами компьютера только через пространства памяти и портов ввода-вывода, а также аппаратные прерывания, то защищать нужно эти три типа ресурсов. Самую сложную защиту имеет память. Операционная система выделяет каждому процессу области памяти – *сегменты* – различного назначения и с разными правами доступа. Из одних сегментов можно только читать данные, в другие возможна и запись. Для программного кода выделяются специальные сегменты, инструкции могут выбираться и исполняться только из них.

Процессору «безразлично» содержимое ячейки памяти, на которую передалось управление, – он всегда пытается трактовать ее как код инструкции или префикс). Если ошибочно управление передалось на область данных, то дальнейшее поведение процессора непредсказуемо. Защита не позволяет передать управление на сегмент данных – сработает исключение защиты, которое обрабатывается операционной системой, и ошибочный процесс будет принудительно завершен. Чтобы выдержать принцип хранимости программы, на время ее загрузки в память или программной модификации ту же область объявляют сегментом данных, в который разрешена запись. Система защиты может полностью контролировать распределение памяти, генерируя исключения в случаях различных нарушений. Эффективность защиты (устойчивость компьютера к ошибкам) в значительной мере определяется предусмотрительностью разработчиков операционной системы.

Чем сложнее программа и больше объем обрабатываемых ею данных, тем больше ее потребности в памяти. В первых процессорах семейства память предоставлялась в виде сегментов с размером по 64 Кбайт, а суммар-

ный объем программно адресуемой памяти не превышал значения в 1 Мбайт. Архитектура PC ограничивала размер оперативной памяти объемом в 640 Кбайт, начиная с нулевых адресов. Эта область называется *стандартной памятью* (conventional memory), и для прикладных программ из нее остается доступной область порядка 400–550 Кбайт (остальное «съедает» операционная система вместе с разными драйверами). Потребности решаемых задач довольно быстро переросли эти ограничения, и в процессоры ввели средства организации *виртуальной памяти.* Впервые они появились в 80286, но удобный для употребления вид приняли только в 32-разрядных процессорах (80386 и выше). Во-первых, было снято ограничение на 64 Кбайтный размер сегмента – теперь любой сегмент может иметь почти произвольный размер до 4 Гбайт. Во-вторых, был введен механизм страничной переадресации памяти (paging). Теперь любая страница (область фиксированного размера) виртуальной логической памяти (адресуемой программой в пределах выделенных ей сегментов) может отображаться на любую область физической памяти (реально установленной оперативной). Отображение поддерживается с помощью специальных таблиц страничной переадресации, в которых кроме связи адресов есть указание на присутствие страницы в физической памяти на данный момент времени. Теперь страница памяти, не нужная процессору в данный момент времени, может быть выгружена на устройство хранения (диск). На ее место можно загрузить нужную страницу. Заявку на загрузку нужной страницы делает сам процессор, без каких-либо усилий выполняемой программы: если программе потребовалась ячейка виртуальной памяти из страницы, образа которой сейчас нет в физической памяти, вырабатывается специальное исключение. Обработчик этого исключения (это часть ОС) найдет свободную физическую страницу (выгрузив на диск ту, которая, по его мнению, пока не нужна), «подкачает» на нее с диска требуемую информацию и вернет управление процессу, прерванному исключением. Этот процесс никто «не заметит» (кроме некоторой задержки выполнения инструкций). Таким образом, в распоряжение всех процессов, исполняемых на компьютере псевдопараллельно, предоставляется виртуальная оперативная память, размер которой ограничен суммой объема физической оперативной памяти и областью дисковой памяти, выделенной для подкачки страниц. Память может быть логически организована в виде одного или множества сегментов переменной длины (в реальном режиме – фиксированный).

### 3.3.6. Регистры дескриптора сегмента

Дескрипторы сегментов – это специальные указатели, определяющие расположение сегмента в памяти.

Регистры дескриптора сегмента невидимы для программиста, однако их содержание очень полезно знать. В x86 регистр дескриптора невидимый для программиста соотнесен с каждым видимым регистром селектора.

Каждый из них содержит 32-битовый базовый адрес сегмента, его границу (предел) и другие необходимые признаки сегмента. Когда адрес сегмента загружается в сегментный регистр, ассоциативный (соотнесенный) регистр дескриптора автоматически модифицируется в соответствии с новой информацией. В режиме реальной адресации только базовый адрес модифицируется напрямую, путем сдвига его значения на четыре разряда влево, поскольку максимальная граница и признаки сегмента фиксированы. В защищенном режиме базовый адрес, граница, все признаки модифицируются содержимым регистра дескриптора сегмента, индексированного селектором. Каждый раз, когда происходит ссылка на ячейку памяти, регистр дескриптора сегмента автоматически вовлекается со ссылкой на ячейку памяти. 32-битовый базовый адрес сегмента становится компонентом вычисления линейного адреса, 32-битовое значение границы используется для операций контроля границы, а признаки проверяются на соответствие типу ссылки на ячейку памяти, которая запрашивается.

| СЕГМЕНТНЫЕ РЕГИСТРЫ | | РЕГИСТРЫ ДЕСКРИПТОРА (загружаются автоматически) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 0 | | физический базовый адрес | граница сегмента | другие атрибуты сегмента из описателя | | | | | | | | | |
| СЕЛЕКТОР | CS | | | | | | | | | | – | | |
| СЕЛЕКТОР | SS | | | | | | | | | – | – | – | |
| СЕЛЕКТОР | DS | | | | | | | | | – | – | – | |
| СЕЛЕКТОР | ES | | | | | | | | | – | – | – | |
| СЕЛЕКТОР | FS | | | | | | | | | – | – | – | |
| СЕЛЕКТОР | GS | | | | | | | | | – | – | – | |

***Рис. 3.3.2***

### 3.3.7. Системные регистры

Дополнительные регистры, введенные в микропроцессор, связаны с работой в защищенном режиме:

1. CR0-CR4 – 32-разрядные управляющие регистры;
2. LDTR – регистр локальной таблицы дескрипторов;
3. GDTR – регистр глобальной таблицы дескрипторов;
4. IDTR – регистр дескрипторной таблицы прерываний;
5. TR – регистр задачи;
6. DR0-DR7 – отладочные регистры;
7. TR3-TR7 – тестовые регистры.

Регистры CR0-CR3 предназначены для общего управления системой. Регистры управления доступны только программам с уровнем привилегий 0.

Регистр CR0 содержит набор бит, которые управляют режимами работы микропроцессора и отражают его состояние глобально, независимо от конкретных выполняющихся задач (таблица 3.3.2).

*Таблица 3.3.2*

**Назначение битов регистра CR0**

| Номер бита в CR0 | Название и назначение | Мнемоника |
|---|---|---|
| 0 | Protection Enable – разрешение защищенного режима. Если PE=PG=0, то процессор работает в реальном режиме. | PE |
| 1 | Monitor Soprocessor – слежение за сопроцессором. Этот бит используется совместно с битом TS. Если MP=TS=1, то при выполнении команды WAIT генерируется особая ситуация 7 – «сопроцессор отсутствует». | MP |
| 2 | Emulate Soprocessor – эмуляция сопроцессора. Если EM=1, то генерируется особая ситуация 7 – «сопроцессор отсутствует» и все команды сопроцессора эмулируются. При выполнении команд MMX этот флаг должен быть сброшен (EM=0). | EM |
| 3 | Task Switcher – переключатель задач. Этот бит устанавливается при каждом переключении задач и вызывает особую ситуацию 7 при поступлении команд сопроцессора, MMX/3DNow! и SIMD. Это нужно, чтобы предотвратить выполнение этих команд над данными, которые остались в оперативной памяти от другой задачи. | TS |
| 4 | Processor Extension Type – тип сопроцессора. Если ET=1, значит, используется 32-битный протокол FPU, если ET=0, используется 16-битный протокол FPU. | ET |
| 5 | Numeric Error – ошибка сопроцессора. Если NE=0, то обработка исключений сопроцессора происходит в стиле MS DOS (через вызов внешнего прерывания). Если NE=1, то действует внутренний механизм обработки исключений при поступлении команд сопроцессора, MMX/3DNow! и SIMD. | NE |
| 16 | Write Protection – защита записи. Если WP=1, то происходит защита записи от записи супервизором операционной системы страниц пользовательского уровня. В противном случае, такая защита снимается. | WP |
| 18 | Alignment Mask – маска выравнивания. Контроль выравнивания операндов в оперативной памяти выполняется, если AM=AC=1 IOPL=3 i386 для i386 AM=0. | AM |
| 20 | No Write – запрет сквозной записи. Работает вместе с битом CD и предназначен для управления внутренней кэш-памятью. | NW |
| 30 | Cash Data – разрешение работы внутренней кэш-памяти (L1, L2) CD=0. | CD |
| 31 | Paging Enable – включение режима страничной адресации (PG=1). | PG |

Функции регистра CR1 пока не определены. Он зарезервирован для будущего использования.

20 старших бит (биты 31-12) регистра CR3 содержат смещение каталога страниц в памяти (используется для преобразования логического адреса в физический адрес), биты 11-5 и 2-0 равны нулю, назначение остальных битов в таблице 3.3.3.

*Таблица 3.3.3*

**Назначение битов регистра CR3**

| Номер бита в CR3 | Название и назначение | Мнемоника |
|---|---|---|
| 3 | Page level Write Transparent – бит, управляющий сквозной записью страниц. Используется для управления кэшированием текущего каталога страниц, если установлен режим страничной адресации (бит PG=1 в регистре CR0). Если PWT=1, тогда обновление текущего каталога страниц происходит методом сквозной записи (write-through), если PWT=0 – методом обратной записи (write-back). | PWT |
| 4 | Page level Cash Disable – бит, контролирующий кэширование каталога страниц. Если PCD=1, то кэширование не происходит. Если PCD=0 – каталог страниц может кэшироваться. Этот флаг влияет только на L1 и L2 (внутренние кэши процессора). Процессор игнорирует этот флаг, если страничная адресация отключена (флаг PG в CR0 сброшен) или если вообще отключен кэш (флаг CD в CR0). | PCD |

Регистр глобальной таблицы дескрипторов содержит 32-битовый базовый адрес глобальной дескрипторной таблицы (начальный адрес системной области памяти, в которой хранятся общедоступные селекторы, описывающие глобальные сегменты памяти, обычно такие сегменты относятся к операционной системе) и 16-битовое значение предела, представляющее собой размер в байтах таблицы GDT.

Регистр дескрипторной таблицы прерываний хранит 32-битовый базовый адрес дескрипторной таблицы прерываний микропроцессора IDT (системной области памяти, в которой размещается таблица прерываний) и 16-битовое значение предела, представляющее собой размер в байтах таблицы IDT.

16-разрядный регистр локальной таблицы дескрипторов содержит селектор, описывающий для данной задачи таблицы дескрипторов. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT.

16-разрядный регистр задачи хранит селектор сегмента, то есть указатель на дескриптор в таблице GDT, в котором записывается информация о текущей задаче, выполняемой микропроцессором. Этот дескриптор опи-

сывает текущий сегмент состояния задачи (TSS – Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит *контекст* (текущее состояние) задачи. Основное назначение сегментов TSS – сохранять текущее состояние в момент переключения на другую задачу.

Регистры GDTR, IDTR 48-разрядные и хранят физические адреса этих таблиц, которые непосредственно выдаются на шину адреса микропроцессора.

32-разрядные отладочные регистры (Debug Registers) DR0-DR7: DR0-DR3 (Linear Breakpoint Address 0..3) содержат 32-битные линейные адреса точек прерывания, а DR4-DR7 устанавливают, что должно произойти при достижении точки прерывания. Регистр DR6 (Breakpoint Status) – регистр состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызывали возникновение последнего исключения с номером 1. Регистр DR7 (Breakpoint Control) – регистр управления отладкой. В этом регистре для каждого из четырех регистров контрольных точек отладки DR0-DR3 имеются поля, с помощью которых можно уточнить условия, при которых следует сгенерировать прерывание.

Тестовые регистры TR3-TR7 (Test Registers) используются для контроля постраничной системы распределения памяти операционной системы: TR3 – регистр данных внутреннего кэша, TR4 – тестовый регистр состояния кэша, TR5 – управляющий регистр тестирования кэша, TR6 (Test Control) – управляющий регистр для теста кэширования страниц, TR7 (Test Status) – регистр данных для теста кэширования страниц.

### 3.3.8. Регистр флагов (RFLAGS/EFLAGS)

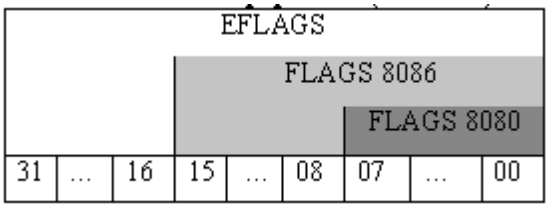



***Рис. 3.3.3***

Регистр RFLAGS/EFLAGS используется для получения информации о результатах выполнения команд и влияет на состояние самого микропроцессора. 64/32-битовый регистр RFLAGS/EFLAGS содержит информа-цию, которая используется, скорее побитно, нежели как 64/32-битное число. Старшие 32 бита RFLAGS равны нулю. Младшие 16 разрядов регистра RFLAGS/EFLAGS, для совместимости со старыми программами, полностью аналогичны регистру FLAGS i8086, 8 младших разрядов которого, в свою очередь, аналогичны регистру FLAGS i8080.

Регистр RFLAGS/EFLAGS представляет собой набор флагов, устанавливаемых или сбрасываемых по результатам выполняемых команд. Флаг – это переменная длиной 1 бит, используемая в командах условного перехода. Если значение этой переменной равно 1, то считается, что флаг установлен, если 0 – сброшен.

Флаги делятся на:

- 8 флагов состояния (NT, IOPL, SF, ZF, AF, PF, CF). Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра RFLAGS/EFLAGS отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм. Для работы с флагом CF существуют специальные команды CLC (очистить флаг CF), STC (установить флаг CF), CMC (инвертировать флаг CF);
- флаг направления (DF) используется цепочечными командами. Значение этого флага определяет направление поэлементной обработки в этих операциях: от начала строки к концу (DF=0), либо наоборот, от конца строки к ее началу (DF=1). Для работы с флагом DF существуют специальные команды CLD (очистить флаг DF), STD (установить флаг DF). Применение этих команд позволяет привести флаг DF в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;
- 5 системных флагов (FT, IF, RF, VM, AC, VIP, VIF, ID), управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы. Для работы с флагом IF существуют специальные команды CLI (очистить флаг IF), STI (установить флаг IF);
- неопределенные биты (XX) являются зарезервированными, то есть в настоящий момент они не имеют значения, однако могут быть использованы для специальных целей в последующих версиях микропроцессора либо уже используются в недокументированных целях. Поэтому программа не должна как-либо использовать ни один зарезервированный бит. Хотя, чем черт не шутит! Если у вас есть старенький компьютер и если Вам его не жалко, можно и поэкспериментировать.

### Флаги состояния

Из флагов состояния программиста, создающего программы на языке ассемблера, в первую очередь интересуют: флаг нуля, флаг переноса и флаг знака.

- CF флаг переноса (**C**arry **F**lag) устанавливается при переносе или займе старшего бита в арифметических операциях, в остальных случаях сбрасывается.
- PF флаг паритета (**P**arity **F**lag) устанавливается, если 8 младших разрядов результата содержат четное число единиц; в противном случае сбрасывается.

- AF вспомогательный флаг переноса (**Auxiliary carry Flag**). Используется командами работающими с BCD числами (AAA, AAS, AAD, AAM, DAA, DAS). Устанавливается при переносе из 3-го бита в 4-й в результате сложения, или при займе из 4-го бита в 3-й в результате вычитания, в остальных случаях сбрасывается.
- ZF флаг нуля (**Zero Flag**) устанавливается в случае получения нулевого результата при выполнении очередной команды и сбрасывается при остальных ненулевых значениях.
- SF флаг знака (**Sign Flag**) устанавливается при единичном значении старшего бита результата – признак отрицательного числа.
- TF флаг трассировки (**Trace Flag**) помогает отлаживать программы. Этот флаг не устанавливается в результате работы микропроцессора, а устанавливается программой с помощью специальной команды. Его также называют флагом трассировки, флагом пошаговой работы или флагом ловушки. Используется преимущественно для осуществления пошагового режима работы. Когда флаг TF установлен, микропроцессор x86 автоматически вырабатывает сигнал внутреннего прерывания INT 1h после выполнения каждой команды. Это обеспечивает удобство проверки программ, написанных на машинном коде, поскольку они при этом выполняются команда за командой. Адрес сервисной процедуры организации пошагового режима должен быть определен в абсолютных адресах от 00004h до 00007h. При генерации прерывания INT 1h микропроцессор x86 автоматически загружает в стек содержимое регистра флагов (с единичным флагом ловушки), после чего обнуляет флаги ловушки и прерываний. Таким образом, микропроцессор x86 не переходит в пошаговый режим пока выполняется сама процедура обслуживания прерывания, которая реализует различные диагностические операции. Процедура обслуживания прерывания INT 1h может включать в себя отображение на дисплее такой информации, как содержимое регистров и определенной области памяти сразу после выполнения команды. Последняя машинная команда в процедуре обслуживания – команда IRET (возврат из прерывания). Этим восстанавливается прежнее единичное состояние флага TF, и по завершении следующей команды вновь генерируется прерывание INT 1h.

  Флаг ловушки TF занимает восьмой разряд в регистре флагов. Для установки флага TF в ноль или единицу можно использовать команду POPF.

  Для противодействия просмотру фрагментов программ под отладчиком подменялась процедура организации пошагового режима, которая выводила на экран не тот фрагмент программы, которую исследовал хакер, а какой-либо другой.
- IF флаг прерываний (**Interrupt Flag**) управляет внешними прерывания. Пока флаг прерываний сброшен в 0, никакие внешние прерывания не будут обрабатываться микропроцессором (за исключением немаскируе-

мых). Когда он установлен в 1, будет производиться обработка любых возникающих прерываний.

- DF флаг направления (**Direct Flag**) устанавливает направление обработки строк.
- OF флаг переполнения (**Overflow Flag**) используется для фиксирования факта потери значащего бита при арифметических операциях, устанавливается при переносе или займе старшего знакового бита в арифметических операциях, в остальных случаях сбрасывается.
- IOPL флаг уровня привилегий ввода /вывода (**Input/Output Privilege Level**). Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода/вывода в зависимости от привилегированности задачи. Уровень привилегий ввода/вывода (IOPL) указывает максимальное значение *текущего уровня привилегий* (CPL – current privilege level). Для максимально допустимого значения CPL при выполнении команд ввода/вывода без генерирования прерывания по #13 исключению он также указывает максимальное значение CPL, позволяющее изменить бит IF, когда новые значения загружаются из стека в регистры FLAGS или EFLAGS. Команды POPF и IRET могут изменять поле IOPL, когда выполняются при CPL=0. Операции включения задач всегда изменяют поле IOPL, когда новый образ флага загружается из сегмента состояния задачи.
- NT флаг вложенности задачи (**Nested Task**). Данный флаг используется в защищенном режиме. NT устанавливается, чтобы показать, что выполнение данной задачи вложено в пределах другой задачи. Если он установлен, то сегмент состояния текущей вложенной задачи имеет достоверную обратную связь с сегментом состояния предыдущей задачи. Данный бит устанавливается или сбрасывается командами, передающими управление другим задачам. Значение NT в EFLAGS проверяется командой IRET. Чтобы установить NT следует выполнять внутризадачное или внешнезадачное возвращение. Команды POPF или IRET будут оказывать воздействие на установку данного бита согласно образу EFLAGS на любом уровне привилегий.
- RF флаг возобновления (**Resumption Flag**) используется при пошаговом режиме или совместно с точками прерываний регистров отладки. Он проверяется на границе команды, перед обработкой точки останова. Если RF установлен, то любая ошибка отладки будет на следующей команде проигнорирована. RF автоматически сбрасывается при успешном окончании каждой команды (ошибки не сигнализируются), кроме команд IRET, POPF, JMP, CALL, ENTER, которые вызывают включение задачи. Эти команды устанавливают RF в соответствии с определенным образом памяти.

  *Например:* В конце обслуживания программы прерываний команда IRET может вытолкнуть образ RFLAGS/EFLAGS, имеющего RF

установленным и возобновляет выполнение программы в адресе точки прерывания без генерирования другого прерывания в том же месте.

*Таблица 3.3.4*

| Номер бита в RFLAGS | Название или содержание | Мнемоника | Статус флага |
|---|---|---|---|
| 0 | Флаг переноса | CF | состояние |
| 1 | = 1 | XX | резервный |
| 2 | Флаг паритета | PF | состояние |
| 3 | = 0 | XX | резервный |
| 4 | Вспомогательный флаг переноса | AF | состояние |
| 5 | = 0 | XX | резервный |
| 6 | Флаг нуля | ZF | состояние |
| 7 | Флаг знака | SF | состояние |
| 8 | Флаг прослеживания (трассировки) | TF | системный |
| 9 | Флаг разрешения прерываний | IF | системный |
| 10 | Флаг направления | DF | управление |
| 11 | Флаг переполнения | OF | состояние |
| 12 | Уровень привилегий ввода /вывода | IOPL | состояние |
| 13 | | | |
| 14 | Флаг вложенности задачи | NT | состояние |
| 15 | = 0 | XX | резервный |
| 16 | Флаг возобновления | RF | системный |
| 17 | Флаг виртуального 8086 | VM | системный |
| 18 | Флаг контроля выравнивания | AC | системный |
| 19 | Флаг виртуального прерывания | VIF | системный |
| 20 | Ожидание виртуального прерывания | VIP | системный |
| 21 | Флаг идентификации | ID | системный |
| 22..63 | = 0 | XX | резервный |

- VM флаг виртуального режима (**V**irtual 8086 **M**ode) обеспечивает виртуальный 8086 режим в пределах защищенного. Если он установлен в то время, когда микропроцессор находится в защищенном режиме, 8086 включается в виртуальную 8086 операцию, манипулируя загрузкой сегментов, как это делает 8086, генерируя 13 прерывание привилегированных операционных кодов. Бит VM может быть установлен в защищенном режиме командой IRET, если текущий привилегированный уровень равен 0, и путем включения задач на любом уровне привилегий. Бит VM не подчиняется действию команды POPF. PUSHF всегда посылает 0 в этот разряд, даже, если работает в виртуальном 8086 режиме. Образ EFLAGS, сохраненный в стеке во время обработки прерывания или во время включения задачи, будет содержать единицу в этом бите, если прерывание обрабатывалось как виртуальная 8086 задача.
- AC контроль выравнивания (**A**lignment **C**ontrol). Флаг контроля используется совместно с битом AM регистра CR0 для отслеживания особых ситуаций, связанных с выравниванием операндов при обращении к

оперативной памяти. Особая ситуация контроля выравнивания генерируется только, если уровень привилегий IOPTL=3.

- VIF виртуальное прерывание (**V**irtual **I**nterruption **F**lag). Виртуальное подобие флага IF. Флаг VIF применяется совместно с флагом VIP для нормального функционирования устаревшего программного обеспечения, использующего команды управления маскируемыми прерываниями.
- VIP флаг ожидания виртуального прерывания (**V**irtual **I**NTERRU**P**TION).
- ID флаг идентификации (**ID**ENTIFICATION). Проверка способности программы поддерживать команду идентификации процессора CPUID.

Не все операции устанавливают флажки. За приблизительное правило можно принять, что:

а) арифметические операции действуют на все флажки,
б) логические операции (кроме операции NOT) сбрасывают флажки переноса и переполнения (OF и CF) и действуют на все другие флажки,
в) операции приращения и уменьшения на 1 (команды INC и DEC) не действуют на флажок переноса и действуют на все другие флажки (нельзя использовать флажок переноса для контроля переполнения при операции уменьшение на 1, но можно использовать флаг знака).

*Не изменяют* флажки:

а) все операции перемещения (MOV, MOVZX, MOVSX, LEA, LDS, LES, LFS, LGS, LSS, XCHG, BSWAP),
б) все операции с портами (IN, OUT, INS, OUTS),
в) все команды перехода (JMP, Jcc, JCXZ, JECXZ),
г) все команды преобразования (CBW, CWD, CWDE, CDQ),
д) все строковые команды (кроме SCAS и CMPS),
е) все операции со стеком (кроме POPF).

### 3.3.9. Указатель команд

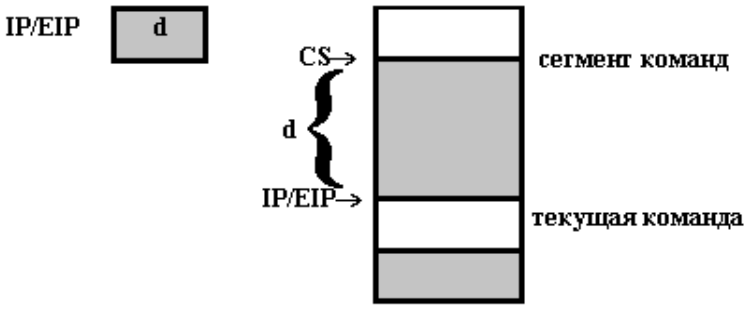



*Рис. 3.3.4*

В регистре IP/EIP/RIP (instruction pointer, указатель команд) всегда находится адрес команды, которая должна быть выполнена следующей. Более точно, в IP/EIP/RIP находится адрес этой команды, отсчитанный от начала сегмента команд, на начало которого указывает регистр CS. Поэтому абсолютный адрес этой

команды определяется парой регистров CS и IP/EIP. За исключением случаев перехода, последовательно выполняемые команды в памяти следуют друг за другом, содержимое регистра IP/EIP/RIP увеличивается после выполнения каждой команды.

В ранних микропроцессорах (до i8086) выборка команд осуществлялась следующим образом: когда микропроцессор был готов к выполнению следующей команды, он посылал содержимое IP по адресной шине во внешнюю память, считывал в соответствующей адресу ячейки памяти 1 байт данных и по шине данных пересылал его в регистр IP. Затем микропроцессор действовал в соответствии с командой и в процессе ее выполнения мог один или несколько раз обращаться к внешней памяти. Одна машинная команда может иметь длину более одного байта. В зависимости от длины команды (для микропроцессоров младше i80386 от 1 до 6 байт, для i80386 и старше от 1 до 15 байт) содержимое увеличивается на 1, 2 или более байт, если это не команда перехода. (При выполнении команды перехода содержимое CS и IP/EIP/RIP может измениться на любую величину.)

*Очередь команд*. Для увеличения скорости выполнения программ в микропроцессоре i8086 регистр IP был дополнен 6-байтной очередью команд, организованной по принципу FIFO («первым пришел – первым ушел»). Очередь команд непрерывно заполняется только тогда, когда системная шина не требуется для других операций. Если системная шина занята – команды выбираются из очереди команд.

*Конвейеризация вычислений*. С появлением микропроцессора i80486 для еще большего увеличения скорости выполнения программ очередь команд была дополнена конвейером. *Конвейер* – специальное устройство, реализующее такой метод обработки команд внутри микропроцессора, при котором исполнение команды разбивается на несколько этапов. i80486 имел пятиступенчатый конвейер. Соответствующие пять этапов включали:

- выборку команды из очереди команд;
- декодирование команды;
- генерацию адреса, при котором определяются адреса операндов в памяти;
- выполнение операции с помощью АЛУ;
- запись результата (куда будет записан результат, зависит от алгоритма работы конкретной машиной команды).

Таким образом, на стадии выполнения каждая машинная команда как бы разбивается на более элементарные операции. Преимущество такого подхода в том, что очередная команда после ее выборки попадет в блок декодирования. Таким образом, блок выборки свободен и может выбрать следующую команду. В результате на конвейере могут находиться в различной стадии выполнения пять команд. Скорость вычисления при этом существенно возрастает. Микропроцессоры, имеющие один конвейер, называются *скалярными*. Pentium имеет два конвейера, Pentium-II – три,

потому эти микропроцессоры называются *суперскалярными*. Микропроцессоры Pentium-III и Pentium-4 называют *гиперконвеерными*, длина конвейера Pentium-III – 10, а у Pentium-4 – 20 ступеней.

*Предсказание правильного адреса перехода*. Под переходом понимается запланированное алгоритмом изменение последовательного характера выполнения программы. Как показывает статистика, типичная программа на каждые 6-8 команд содержит 1 команду перехода. Последствия этого предсказать не сложно: при наличии команды перехода через каждые 6-8 команд конвейер и очередь команд нужно очищать и заполнять заново в соответствии с новым адресом перехода. Все преимущества конвейеризации теряются.

Поэтому в архитектуру Pentium был введен *блок предсказания переходов*. Суть этого метода заключается в следующем. Pentium имеет буфер адресов перехода, который хранит информацию о последних 256 переходах. Если некоторая команда управляет ветвлением, то в буфере запоминаются эта команда, адрес перехода и предположение о том, какая ветвь программы будет выполнена следующей. Почти в любой программе имеются циклы, в ходе выполнения которых периодически необходимо принимать решение либо о выходе из цикла, либо о переходе на его начало. Специальный блок предсказания адреса перехода прогнозирует, какое решение будет принято программой. При этом он основывается на предположении, что ветвь, которая была пройдена, будет использоваться снова, и загружает соответствующую команду перехода на конвейер. В случае если это предсказание верно, переход осуществляется без задержки. Для того чтобы судить об эффективности этого нововведения, достаточно отметить, что вероятность правильного предсказания составляет 80 %.

## 3.4. Система команд

Описание любого микропроцессора выглядит иначе, чем описание большинства других микросхем, отчасти из-за присутствия в нем обширного раздела, посвященного «системе команд». Для использования микропроцессора важно понимать, что «делают» команды: как обращаться к памяти («типы адресации»), к регистрам, куда попадают результаты арифметических операций, как устроить условное ветвление (переходы) и т.д. Каждая команда микропроцессора x86 состоит из одного, двух или более байт, причем первый байт – это опкод команды. Опкод определяет природу команды; по опкоду микропроцессор определяет, нужны ли дополнительные байты, и, если да, получает их в последующих циклах. Поскольку опкод состоит из 8 бит, может существовать 256 разных опкодов. Ограничение в 256 команд было обойдено, так как некоторые опкоды (0FEh, 0FFh и другие) служат шлюзами к следующим таблицам кодов. В приложении представлен полный список команд микропроцессора x86.

В таблице команды даны не в шестнадцатеричном машинном представлении (машинном коде), а в виде мнемокодов языка ассемблера, поскольку микропроцессор x86 обычно программируют на этом более удобочитаемом языке, обозначая адреса и числа легко запоминаемыми именами вместо чисел, с которыми в действительности оперирует микропроцессор. После того как программа написана на языке ассемблера, стандартная программа, известная также под названием «ассемблер», преобразует мнемокоды и имена в числа. В связи с этим язык ассемблера называют «входным языком» в отличии от числового «объектного языка», который микропроцессор x86 использует непосредственно. Таким образом, программа ассемблер переводит на объектный язык программы, составленные на входном языке. В процессе этого перевода одна команда языка ассемблер может быть преобразована в 1, 2 или более байт объектного языка в зависимости от конкретной команды.

Последующие главы погрузят Вас в изучение системы команд микропроцессора x86, поскольку это единственный способ составить представление о его работе.

## 3.5. Система прерываний

*Прерывания* – это аппаратный механизм, который заставляет микропроцессор прервать выполнение текущей задачи и заняться обработкой внешнего события. Первоначально прерывания были разработаны для повышения эффективности планирования использования микропроцессора. Одним из мотивов было желание предоставить «интеллектуальным» устройствам возможность производить операции ввода-вывода независимо от микропроцессора. Устройству необходимо только сигнализировать с помощью прерывания микропроцессору о том, что передача данных завершена. Другой мотив возник из происходящих внутри микропроцессора событий, а именно случаев ошибок (как, например, слишком большое целое число, деление на ноль, выход за границы памяти и т.п.), а также связанных с операционной системой отслеживающих механизмов. Последний мотив возник в многопроцессорных системах, в которых каждый микропроцессор должен сигнализировать другим о доступе к разделенным ресурсам.

Прерывание – это сигнал микропроцессору, вынуждающий его отвлечь свое внимание от текущей деятельности. Ценность аппарата прерываний заключается в том, что микропроцессор может автоматически реагировать на внешние по отношению к системе ситуации, а также на ситуации, возникающие внутри него самого. Аппарат прерываний может включать в себя несколько типов прерываний. Наиболее общими, конечно, являются прерывания, генерируемые периферийными устройствами, требующими обслуживания после завершения операции ввода-вывода.

Другой источник прерываний – это устройства управления памятью, которое может сигнализировать об обращении к виртуальной странице памяти, отсутствующей в оперативной памяти, или об ошибочной адресации. Внутри микропроцессора прерывания могут генерироваться в случае арифметических ошибок. И, наконец, посредством выполнения специальной команды может быть сгенерировано программное прерывание.

В результате прерывания происходит переход к программе обработки прерываний. Прежде чем приступить к анализу причины прерывания, эта программа должна сохранить текущее состояние микропроцессора. Когда выполнение программы обработки прерываний завершено, управление должно быть снова передано прерванному процессу, причем таким образом, чтобы вклинившаяся работа программы обработки прерываний никак не отразилась на выполняемых им операциях.

Существуют три основных типа механизма прерываний: прерывание, вызванное сбросом или запуском микропроцессора, прерывания по вызову и векторные прерывания.

### 3.5.1. Внешние прерывания

Микропроцессор x86 имеет три линии прерываний – RESET, NMI и INTR, по которым внешние устройства могут передавать свои запросы на обслуживание со стороны микропроцессора.

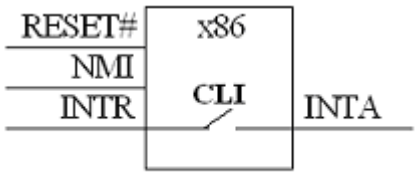



***Рис. 3.5.1***

*Запуск микропроцессора x86*. Аппаратный сброс выполняется процессором при включении питания или при появлении запроса в линии RESET#. Микропроцессор x86 прекращает выполнение инструкций и перестает управлять системной шиной. Процессору устанавливается коэффициент умножения тактовой частоты, режим (WB/WT) работы кэша, роль процессора в многопроцессорных системах, способ подачи сигналов прерываний (для процессоров, имеющих APIC) и некоторые другие параметры. Сброс переводит процессор в реальный режим и устанавливает ряд регистров в определенное состояние. Микропроцессор производит следующие действия:

- устанавливает флаг IF в нулевое состояние. Это приводит к невозможности выполнения маскируемых прерываний;
- аннулирует строки кэш-памяти, буферов трансляции (TLB) и таблиц переходов (BTB);

- обнуляет регистры сегментов DS, ES и SS;
- обнуляет указатель команд;
- засылает число 0FFFFh в регистр сегмента команд CS.

Микропроцессор x86 начинает работу с обращения к ячейке с адресом 0FFFFFFF0h. Эта ячейка памяти содержит команду JMP, которая передает управление процедуре инициализации, запускающей компьютер.

*Векторы прерываний* служат для идентификации процедур, необходимых для обслуживания требования прерывания. Каждому внешнему требованию на прерывание может быть поставлен в соответствие код прерывания в пределах от 0 до 255. В микропроцессоре x86 существует 256 векторов прерываний – по одному вектору на каждый тип прерывания. Таблица векторов прерываний занимает 1024 младших байта памяти – ячейки с физическими адресами от 00000 до 003FFh – и имеет 256 входов в соответствии с количеством векторов.

Каждый вход таблицы является указателем двойного слова, содержащего начальный адрес процедуры, которая обеспечивает обслуживание требования на прерывание данного типа. Старшее 16-битовое слово каждого входа таблицы содержит перемещаемый адрес процедуры обслуживания внутри сегмента. Так как каждый вход таблицы занимает четыре байта, первая ячейка входа для прерывания данного типа определяется умножением кода типа прерывания на четыре.

*Маскируемые прерывания*. Это внешние требования, поступающие в микропроцессор по линии INTR. С помощью маскируемых прерываний можно засинхронизовать с микропроцессором наибольшее число внешних устройств. При активизации линии INTR микропроцессор совершает различные действия, зависящие от состояния флага прерываний IF. Реализуемая в этот момент машинная команда всегда выполняется до конца, и только после этого начинается обработка запроса на прерывание.

Если флаг прерываний IF=0 (прерывание не возможно), то требование маскируемого прерывания игнорируется, и микропроцессор продолжает выполнять очередную команду текущей программы. Если IF=1 (прерывание разрешено), микропроцессор подтверждает требование прерывания и передает управление той процедуре, которая должна обслужить поступившее требование. Для выполнения требования маскируемого прерывания микропроцессор автоматически выполняет следующую последовательность действий:

1. генерируется сигнал подтверждения внешнего прерывания $\overline{INTA}$. Этот сигнал сообщает внешнему устройству о том что прерывание признано;
2. считывается код прерывания *N*, поступившего на шину данных;
3. в стек включается текущее содержимое FLAGS (EFLAGS), CS и IP (EIP) (именно в таком порядке);

4. флажки IF и TF сбрасываются, что предотвратит возможность выполнения любого вновь поступающего маскированного прерывания и делает невозможным пошаговый режим;
5. содержимое ячеек памяти $4\times N$ и $4\times N+1$ передается в IP(EIP), а содержимое памяти по адресам $4\times N+2$ и $4\times N+3$ в CS.

После выполнения пункта 5 управление передается процедуре обслуживания прерывания, содержащей машинные команды, необходимые для удовлетворения требований маскированного прерывания. Одной из таких команд, содержащихся в процедуре обслуживания, является команда STI (установить прерывание), которая устанавливает флаг IF=1 и тем самым делает возможным выполнение новых требований, поступающих на INTR.

*Немаскируемые прерывания.* Это внешние требования, поступающие в микропроцессор по линии NMI. Обычно ими являются прерывания, сигнализирующие микропроцессору о внешних событиях особой важности (носящих катастрофический характер), таких как отключение питания, сбой памяти и т.п. Немаскируемые прерывания принимаются микропроцессором всегда независимо от состояния флага прерываний IF, поэтому немаскируемые прерывания имеют более высокий приоритет по сравнению с маскируемыми прерываниями. Им присвоен тип 2. Микропроцессор в ответ на требование немаскируемого прерывания выполняет следующие действия:

1. включить в стек текущее содержимое FLAGS (EFLAGS), CS и IP(EIP);
2. сбросить флажки IF и TF;
3. передать содержимое ячеек памяти 00008h и 00009h в IP(EIP), а содержимое памяти по адресам 0000Ah и 0000Bh в CS.

После выполнения пункта 3 управление передается процедуре обслуживания прерывания, содержащей машинные команды, необходимые для удовлетворения требования, поступившего по линии NMI.

### 3.5.2. Внутренние прерывания

Внутренние прерывания обуславливаются прикладными программами, использующими команду INT. Они возникают также при некоторых условиях автоматически по сигналам в самом микропроцессоре, например, при ошибках деления, переполнении и т.п. Внутренние прерывания аналогичны маскируемым прерываниям, требования на которые поступают по линии INTR. Отличие заключается в том, что микропроцессор реагирует на запросы внутренних прерываний независимо от состояния флага IF.

Код типа внутреннего прерывания может быть задан аппаратно либо представлен частью машинной команды. Если тип прерывания кодируется командой, то оно называется ***программным прерыванием***. Последнее яв-

ляется удобным средством проверки процедур прерывания, составленных для обслуживания внешних устройств.

Обычно программные прерывания используются для реализации возможностей, предоставляемых системным программным обеспечением.

***Прерывание из-за ошибки деления.*** Микропроцессор автоматически генерирует прерывания типа 0 немедленно вслед за операциями DIV (целочисленное деление без учета знака) или IDIV (целочисленное деление с учетом знака) при возникновении следующих условий:

- деление на 0;
- если результат занимает больше 8/16/32 или 64 бит при делении соответственно 8/16/32- или 64-разрядных чисел.

Адрес процедуры обслуживания прерывания типа 0 должен определяться ячейками памяти с физическими адресами с 00000h по 00003h.

***Пошаговое прерывание.*** Когда флаг TF установлен, микропроцессор x86 автоматически вырабатывает сигнал внутреннего прерывания INT 1h после выполнения каждой команды. Это обеспечивает удобство проверки программ, написанных на машинном коде, поскольку они при этом выполняются команда за командой. Адрес сервисной процедуры организации пошагового режима должен быть определен в ячейках памяти с физическими адресами от 00004h до 00007h. При генерации прерывания INT 1h микропроцессор x86 автоматически загружает в стек содержимое регистра флагов (с единичным флагом ловушки), после чего обнуляет флаги ловушки и прерываний. Таким образом, микропроцессор x86 не переходит в пошаговый режим пока выполняется сама процедура обслуживания прерывания, которая реализует различные диагностические операции. Процедура обслуживания прерывания INT 1h может включать в себя отображение на дисплее такой информации, как содержимое регистров и определенной области памяти сразу после выполнения команды. Последняя машинная команда в процедуре обслуживания – команда IRET (возврат из прерывания). Этим восстанавливается прежнее единичное состояние флага TF, и по завершении следующей команды вновь генерируется прерывание INT 1h.

***Прерывание в точке.*** Прерывание в заданной точке программы используется для отладки при ее написании и тестировании. Микропроцессор генерирует прерывание типа 3 немедленно по завершении выполнения команды INT 3h. Шестнадцатеричный машинный код этой команды 0CCh используется как команда прерывания в заданной точке программы. Этот код вставляется в любое место программы, где необходимо прервать ее нормальное выполнение, и с помощью процедуры, обслуживающей это прерывание, отобразить критическую информацию, такую как содержимое внутренних регистров микропроцессора и ячеек памяти. Поскольку минимальная смысловая часть команды INT 3h занимает один байт, для использования в различных программных отладчиках для установки точек прерывания (*break point*) машинный код 0CCh может подменить любую

команду, обозначая тем самым точку программы, в которой необходимо осуществить прерывание. Адрес сервисной процедуры организации обработки прерывания в заданной точке программы должен быть определен в ячейках памяти с физическими адресами от 0000Ch до 0000Fh.

***Прерывание, если переполнение.*** Микропроцессор генерирует прерывание типа 4 или INTO если обнаружено, что OF=1. Если команда в программе может в результате своей работы установить флаг переполнения OF (к примеру, арифметические команды), то для обнаружения и обработки такой ситуации можно использовать команду INTO или INT 4h (машинный код 0CEh). Особенности передачи управления и обработки (корректировки) результата зависят от режима работы микропроцессора. Адрес сервисной процедуры организации обработки прерывания по переполнению должен быть определен в ячейках памяти с физическими адресами от 00010h до 00013h.

## Контрольные вопросы и упражнения

1. Какие регистры можно использовать для следующих целей: а) сложения и вычитания, б) подсчета числа циклов, в) для умножения и деления, г) для адресации сегментов, д) индикации нулевого результата, е) адресации выполняемой команды.
2. Какие типы сегментов вы знаете?
3. Чему равен размер сегмента?
4. Если физический адрес равен 5A230h, когда CS=5200h, каким он будет при изменении CS на 7800h?
5. Пусть смещение переменной в сегменте равно 2359h и DS=490Bh. Чему равен физический адрес переменной?
6. Найти состояние флагов AF, SF, ZF, CF, OF и PF после прибавления 62A0h к следующим числам:

   а) 1234h, б) 4321h, в) 0CFA0h, г) 9D60h.

7. Найти состояние флагов SF, ZF, CF, OF и PF после вычитания 4AE0h из следующих чисел:

   а) 1234h, б) 5D90h, в) 9090h, г) EA04h.

8. Преобразовать логический адрес (сегмент:смещение) в физический:

   1) 2397h:0100h; 2) 241Ch:002Bh; 3) 245Ch:0002h; 4) 23A7h:0000h.

ГЛАВА 4

# ЭТАПЫ СОЗДАНИЯ ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Разработка программы включает несколько этапов:

1. подготовка (изменение) исходного текста программы,
2. ассемблирование программы (получение объектного кода),
3. компоновка программы (получение исполняемого файла программы),
4. запуск программы,
5. отладка программы.

Обычно эти этапы циклически повторяются, потому что при нахождении ошибок при ассемблировании, компоновки или отладке приходится вновь возвращаться к первому этапу и изменять текст программы для устранения ошибок.


***Рис. 4.1. Этапы разработки программ на ассемблере***

## 4.1. Подготовка текста программы

Текст программы записывается в один или несколько текстовых файлов. Имена файлов могут быть любыми, но для файлов, содержащих текст программы, принято расширение **.asm**, а для файлов с определением констант и новых типов – расширение **.inc**. Эти файлы являются текстовыми, их можно подготовить с помощью стандартных редакторов текста или с использованием интегрированных сред разработки программ.

### 4.1.1. Использование стандартных редакторов

При использовании стандартных редакторов текста необходимо сохранять редактируемые файлы с текстами программ в виде обыкновенных файлов в формате ASCII, то есть без дополнительных символов форматирования, которые вставляются в текст специализированными редакторами, например Word. Иногда используются специализированные текстовые редакторы для программистов, с разноцветной подсветкой синтаксиса,

автоматической нумерацией строк, всплывающими подсказками и тому подобными «наворотами».

## 4.2. Ассемблирование программы

Подготовленный текст является исходными данными для специальных программ, называемых ассемблерами. Задача ассемблеров – преобразовать текст программы в форму двоичных команд, которые могут быть выполнены микропроцессором. Если обнаружены синтаксические ошибки, то результирующий код создан не будет. Процесс создания исполняемого файла происходит в две стадии:

.asm → .obj → .exe/.dll/.com

На первой стадии (.asm → .obj) из ассемблерного файла путем *компиляции* получаются файлы промежуточного *объектного кода,* имеющего расширение **.obj** (при этом могут использоваться дополнительные inc-файлы). Файл с расширением **.obj** содержит оптимизированный машинный код при условии, что не встретились синтаксические и семантические ошибки. Если в исходном файле с программой на языке ассемблера обнаруживаются ошибки, то программисту выдается список обнаруженных ошибок, в котором ошибки указываются с номером строки, в которой они обнаружены. Программист циклически выполняет действия по редактированию и компиляции до тех пор, пока не будут устранены все ошибки в исходном файле. На этом этапе уже возможно получение готовой программы, но чаще всего в ней не хватает некоторых компонентов. Если компилятор по какой-либо причине (неверно прописан путь к такому файлу или файл отсутствует) не может найти inc-файл, то выдается предупреждение и obj-файл получен не будет.

Ассемблирование, как правило, проходит в два приема. При первом проходе переводятся мнемонические команды, десятеричные числа и символы в соответствующие машинные коды, подсчитывается, сколько какая команда занимает места, обнаруженные имена, введенные пользователем (константы, метки, переменные) их тип и числовое значение записывается в таблицу. В эту же таблицу записывается, с каких адресов начинаются процедуры, адреса меток, адреса начала/конца сегментов и т. д., при втором проходе подставляются адреса начала процедур, заменяются названия меток на адреса.

В результате ассемблирования получается так называемый «объектный файл». В качестве дополнительной возможности ассемблер может создать файл листинга программы.

Обычно для получения файлов объектного кода необходимо выполнить соответствующую программу ассемблера (программы MASM.EXE и ML.EXE фирмы Microsoft и TASM.EXE или TASM32.EXE фирмы

Borland), указав в командной строке имя файла с текстом программы. Например, если у текстового файла с исходным текстом программы название prog.asm.

**ml prog.asm**

или

**tasm prog.asm**

Эта форма вызова является минимально необходимой. Кроме имени текстового файла, необходимо указывать опции ассемблирования. Более подробную информацию об опциях программы ассемблирования следует искать в документации к этим программам.

## 4.3. Компоновка программы

Следующая стадия (.obj → .exe/.dll/.com) называется *линковкой* или *компоновкой* и служит для замещения символьных имен, используемых программистом, на реальные адреса.

Сравните шестнадцатеричное содержимое **OBJ** и **EXE** файла, который у вас получился. В EXE-файле присутствует та же последовательность байтов, что и в OBJ-файле. Но помимо этого еще присутствует: имя ассемблированного файла, версия ассемблера, «имя собственное» сегмента и так далее.

Это «служебная» информация, предназначенная для тех случаев, когда ваш исполнимый файл вы хотите собрать из нескольких. При разработке больших приложений исходный текст состоит, как правило, из нескольких *модулей* (файлов с исходными текстами), потому что хранить все тексты в одном файле неудобно – в них сложно ориентироваться. Каждый модуль по раздельности компилируется в отдельный файл с объектным кодом. В каждом из этих файлов прописаны свои сегменты кода/данных/стека, которые затем надо объединить в одно целое. А исполнимый файл нам нужно получить только один – с единым сегментом кода/данных/стека. Именно это **LINK** и делает: завершает определение адресных ссылок и объединяет, если это требуется, несколько программных модулей в один. И этот один у нас и является исполнимым.

Кроме того, к нашим модулям надо добавить машинный код подпрограмм, реализующих различные стандартные функции (например, вычисляющих математические функции SIN или LN). Такие функции содержатся в библиотеках (файлах со стандартным расширением .LIB), которые либо поставляются вместе с компилятором, либо создаются самостоятельно. Поэтому процесс подготовки обязательно включает в себя этап компоновки, когда определяются все неизвестные при раздельном ассемблировании адреса совместно используемых переменных или функций.

Процесс объединения объектных модулей в один файл осуществляется специальной программой-компоновщиком или *сборщиком*

(программа LINK.EXE фирмы Microsoft и TLINK.EXE фирмы Borland), которая выполняет связывание объектных модулей и машинного кода стандартных функций, находя их в библиотеках, и формирует на выходе работоспособное приложение – *исполнимый код* для конкретной платформы.

Исполнимый код – это законченная программа с расширением COM, DLL или EXE, которую можно запустить на компьютере с установленной операционной системой, для которой эта программа создавалась. Имя исполняемого файла задастся именем первого .OBJ файла**:**

**link prog1.obj prog2.obj**

Содержимое объектного файла анализируется компоновщиком. Он определяет, есть ли в программе *внешние ссылки*, то есть содержит ли программа команды вызовов процедур, находящихся в одной из библиотек *объектных модулей* (link library). Компоновщик находит эти ссылки в объектном файле, копирует необходимые процедуры из библиотек, объединяет их вместе с объектным файлом и создает *исполняемый файл* (executable file). В качестве дополнительных возможностей компоновщик может создать *файл перекрестных ссылок*, содержащих план полученного исполняемого файла.

## 4.4. Загрузка программы

Компонент операционной системы, называемый загрузчиком (loader), считывает данные из исполняемого файла, загружает программу в память и передает управление по адресу точки входа. В результате программа начинает выполняться.

В тех случаях, когда при написании новой программы на языке ассемблера требуются лишь незначительные изменения машинных кодов, иногда быстрее и удобнее внести изменения непосредственно в объектный файл, а не проходить всю цепочку редактирования исходной программы и осуществлять ее повторную трансляцию с внесенными изменениями. Для этого существуют специальные шестнадцатеричные редакторы (типа Hacker Viewer), которые позволяют рассматривать файлы с бинарным (машинным) кодом в виде последовательности ассемблерных команд. Эту же технологию применяют в тех случаях, если исходный текст программы не доступен (*взлом программы*).

## 4.5. Отладка программы

До тех пор пока Вы не наберетесь достаточного опыта в программировании на языке ассемблера, за исключением учебных, тривиальных программ, в отладке будет нуждаться любая ваша программа. Для этого используются различные отладчики (CodeView фирмы Microsoft, Turbo

Debugger фирмы Borland, SoftIce фирмы NuMega, OllyDebug написанный Olech Yuschuk), позволяющие в процессе выполнения программы контролировать значения регистров или переменных, при необходимости изменять их. Можно просматривать содержимое различных участков памяти. Можно выполнять программы по шагам или расставить точки останова (BREAK POINTS), которые вызовут прекращение работы программы и переход управления к отладчику.

## 4.6. Использование интегрированных сред

Для подготовки текста программы на языке ассемблера очень удобно использование интегрированных программных сред. Такие возможности предоставляются практически всеми разработчиками ассемблеров, например: PWD – Programmer Workbench для компиляторов masm, QC – Quick C для компиляторов QuickAssembler фирмы Microsoft, все компиляторы языка C и C++ фирмы Borland (TC и BC). Интегрированные среды позволяют получить быстрый доступ к справочной информации. Можно сразу же ассемблировать и скомпоновать набранный текст, провести его отладку, а затем вновь вернуться к редактированию.

## 4.7. Структура программы

Как вы увидите в следующих главах, язык ассемблера заставляет нас помещать определенное количество строк в качестве заголовка программ, которые мы пишем. Другими словами, нам нужно каждый раз записывать несколько псевдооператоров, которые сообщают языку ассемблера основную информацию. В качестве рекомендации на будущее ниже приведен абсолютный минимум, необходимый для программ, которые вы пишите:

```
.686P
.model flat
.code
star:
;тело вашей программы
ret
.data
;данные вашей программы
end star
```

Разберем ее подробнее. В первой строке .686P – это директива описания типа микропроцессора (может быть еще и .8086, .8087, .186, .286, .287, .386, .387, .486, .586, .mmx с добавлением или без добавления букв P (привилегированные команды) и C или N (непривилегированные команды)). Если не указывать тип микропроцессора, то программа будет сгенерирована в кодах i8086.

*Таблица 4.7.1*

**Директива описания типа микропроцессора**

| Директива | Назначение |
|---|---|
| .8086 | Разрешены инструкции базового процессора i8086 (и идентичные им инструкции процессора i8088). Запрещены инструкции более поздних процессоров. |
| .186 .286<br>.386 .486<br>.586 .686 | Разрешены инструкции соответствующего процессора x86 (x=1,…,6). Запрещены инструкции более поздних процессоров. |
| .187 .287<br>.387 .487<br>.587 | Разрешены инструкции соответствующего сопроцессора x87 наряду с инструкциями процессора x86. Запрещены инструкции более поздних процессоров и сопроцессоров. |
| .286c .386c<br>.486c .586c<br>.686c | Разрешены НЕПРИЛЕГИРОВАННЫЕ инструкции соответствующего процессора x86 и сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров. |
| .286p .386p<br>.486p .586p<br>.686p | Разрешены ВСЕ инструкции соответствующего процессора x86, включая привилегированные команды и инструкции сопроцессора x87. Запрещены инструкции более поздних процессоров и сопроцессоров. |
| .mmx | Разрешены инструкции MMX-расширения. |
| .xmm | Разрешены инструкции XMM-расширения. |
| .K3D | Разрешены инструкции AMD 3D. |

Модель памяти задается директивой .model

Строка *.model flat* говорит, что будет создаваться exe-файл для 32-разрядной операционной системы Windows. *Плоская* (flat) модель памяти 32-разрядной Windows располагает три сегмента (сегмент кода, стека и данных) в едином четырехгигабайтном адресном пространстве, позволяя вообще забыть о существовании сегментов. Но для 16-разрядных приложений MS-DOS и Windows 3.x максимально допустимый размер сегментов составляет всего лишь 64 килобайта, что явно не допустимо для большинства приложений. В *крошечной* (tiny) модели памяти сегмент кода, стека и данных также расположены в едином 64-килобайтном адресном пространстве, но в отличие от плоской модели это адресное пространство чрезвычайно ограничено в размерах, поэтому и код, и стек, и данные более серьезных приложений приходилось размещать в нескольких сегментах (модели памяти small, medium, compact, large, huge, tchuge). В этих моделях памяти, например, для вызова функции недостаточно было знать ее смещение, а требовалось указать еще и сегмент, в котором функция была расположена. Команды передачи управления переходы (*jmp)* и вызовы (*call)* в этих моделях памяти могут быть близкими (*near*) и дальними (*far*). Если вы пишете программы для 32/64-разрядной операционной системы Windows, то о других моделях памяти кроме flat можно забыть со спокойной совестью.

Для адресации четырех гигабайтов виртуальной памяти, выделенной в распоряжение процесса, Windows использует два селектора, один из которых загружается в сегментный регистр *CS*, а другой в регистры *DS*, *ES* и *SS*. Оба селектора ссылаются на один и тот же базовый адрес памяти, равный нулю, и имеют идентичные лимиты, равные четырем гигабайтам. Windows использует еще и регистр *FS*, в который загружается селектор сегмента, содержащего информационный блок потока TIB.

Фактически существует всего один сегмент, вмещающий в себя и код, и данные, и стек процесса. Благодаря этому передача управления коду, расположенному в стеке, осуществляется близким (*near*) вызовом или переходом. Отличия между регионами кода, стека и данных заключаются в атрибутах принадлежащих им страниц страницы кода допускают чтение и исполнение, страницы данных чтение и запись, а страницы стека чтение, запись и исполнение одновременно.

Допустим у нас есть *логический* адрес **0137:00456789h**. Чтобы этот адрес перевести в *линейный* – в селекторе 137 находится соответствующий ему дескриптор в таблице дескрипторов: база = 0, граница = 0FFFFFFFFh, следовательно, линейный адрес равен **0** (база) + **00456789h**. Однако линейный адрес не является *физическим* адресом. Для его получения используется третья ступень – страничная адресация. То есть 20 старших бит линейного адреса используются для выбора 4 Кбайт памяти из каталога страниц, оставшиеся 12 бит представляют смещение внутри полученной страницы (в качестве упражнения рекомендую написать небольшую программу под 32-разрядной Windows, которая будет показывать сегментные регистры, значения дескрипторов для каждого селектора, базу, границу, RPL и т.п). В 32-разрядной Windows и сегмент кода, и сегмент данных и стека приложения имеют одинаковые базу и границу (0 и 0FFFFFFFFh). Это называется плоской (**FLAT**) моделью памяти. Хотя **cs** и **ds** имеют разные значения и дескрипторы, они указывают на одно и то же линейное адресное пространство 0..0FFFFFFFFh. Следовательно, логические адреса cs:12345678 и ds:12345678 совпадают. Есть возможность модифицировать код при помощи **mov byte ptr $+8,21h** (секция кода должна быть помечена как writeable). В данном случае в инструкции **mov** неявно подразумевается **ds:**, в который можно писать. Однако, при попытке сделать **mov cs:xxxxxxxx**, получим исключение (сегментная защита). В сегмент кода писать нельзя, но зато можно писать в сегмент данных, который «совпадает» с сегментом кода, и тем самым модифицировать код. А теперь вспомним про страничную защиту. Именно она используется в Windows, когда Вы задаете атрибуты *секций* PE-файла (**.data**, **.code** и т.д). Собственно, к *сегментам* памяти они не имеют отношения, посему когда речь идет о Win32, не путайте понятия секций PE-файлов и сегментов памяти! Когда Windows грузит PE-файл, она смотрит атрибуты *секций* и соответственно

им устанавливает «защиту» *страниц* памяти, в которые будет загружена секция. Это и есть типа страничная защита.

Помимо этого каждая страница имеет специальный флаг, определяющий уровень привилегий, необходимых для доступа к этой странице. Некоторые страницы, например, те, что принадлежат операционной системе, требуют наличия прав супервизора, которыми обладает только код нулевого кольца. Прикладные программы, исполняющиеся в кольце 3, таких прав не имеют и при попытке обращения к защищенной странице порождают исключение.

*Таблица 4.7.2*

**Директива .model**

| **Модель памяти** | **Количество и размер сегментов** | | **Тип указателя** | | **Описание** |
|---|---|---|---|---|---|
| | **кода** | **данных** | **для кода** | **для данных** | |
| 16-разрядные приложения MS-DOS и Windows 3.x | | | | | |
| Tiny | один, ≤64Kb | | near | near | Код, данные и стек находятся в одном сегменте. Эта модель памяти используется для написания программ типа .COM |
| Small | один, ≤64Kb | один, ≤64Kb | near | near | Код программы находится в одном сегменте. Данные и стек находятся в другом сегменте |
| Medium | несколько, ≤64Kb | один, ≤64Kb | far | near | Код находится в нескольких сегментах, по одному на каждый программный модуль. Данные объединены в один сегмент |
| Compact | один, ≤64Kb | несколько, ≤64Kb | near | far | Код находится в одном сегменте. Данные могут находится в нескольких сегментах. Для ссылки на данные из кода применяются указатели дальнего типа |
| Large | несколько, ≤64Kb | несколько, ≤64Kb | far | far | Код может размещаться в нескольких сегментах, на каждый модуль новый сегмент. Данные также размещаются в нескольких сегментах. Для ссылки на данные из кода применяются указатели дальнего типа. |

| Модель памяти | Количество и размер сегментов | | Тип указателя | | Описание |
|---|---|---|---|---|---|
| | кода | данных | для кода | для данных | |
| Huge | несколько, >64Kb | несколько, >64Kb | huge | huge | Код может размещаться в нескольких сегментах, на каждый модуль новый сегмент. Данные и код имеют тип huge |
| Tchuge | несколько, ≤64Kb | несколько, ≤64Kb | far | far | Также как для модели large, но с иным использованием сегментных регистров |
| 32-разрядная Windows | | | | | |
| Flat | не ограничено | не ограничено | flat | flat | Соответствует варианту модели small, но с использованием 32-разрядной адресации |

В третьей строчке написано *.code*. По этой команде (директиве) будет определен сегмент кода. Это нужно, потому что у реальных программ код (команды) и данные (переменные) должны быть разделены. Если Вы хотите в каком-либо месте программы вставить данные, то пишите .data, а потом уже сами данные (но их можно и не разделять). Я рекомендую вставлять данные между командами *ret* и *end start*. *start:* – это место, где находится точка входа в программу. На языке ассемблера всегда надо помечать место, где находится начало программы (первая команда). Строка *ret* нужна, чтобы выйти из программы – если бы ее не было, то программа бы «завесила» компьютер. Последняя строчка *end start* завершает текст исходной программы.

## 4.8. Пишем первую программу на языке ассемблера

Итак это была теория. А вот это практика. Попробуем написать программу на языке ассемблера самостоятельно. Все что вам нужно это компьютер и набор программ masm32.

Сначала наберем с помощью текстового редактора вот такой файл:

```
.686P
.model flat
include windows.inc
includelib user32.lib
extern _imp__MessageBoxA@16:dword
.code
start: push 0
     push offset sztext
     push offset szText
```

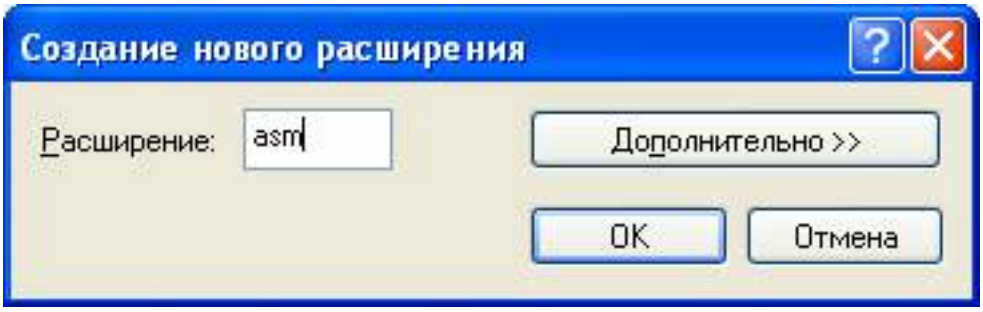
```
      push 0
      call _imp__MessageBoxA@16
      ret
sztext db "Моё первое приложение",0
szText db "Привет!",0
end start
```

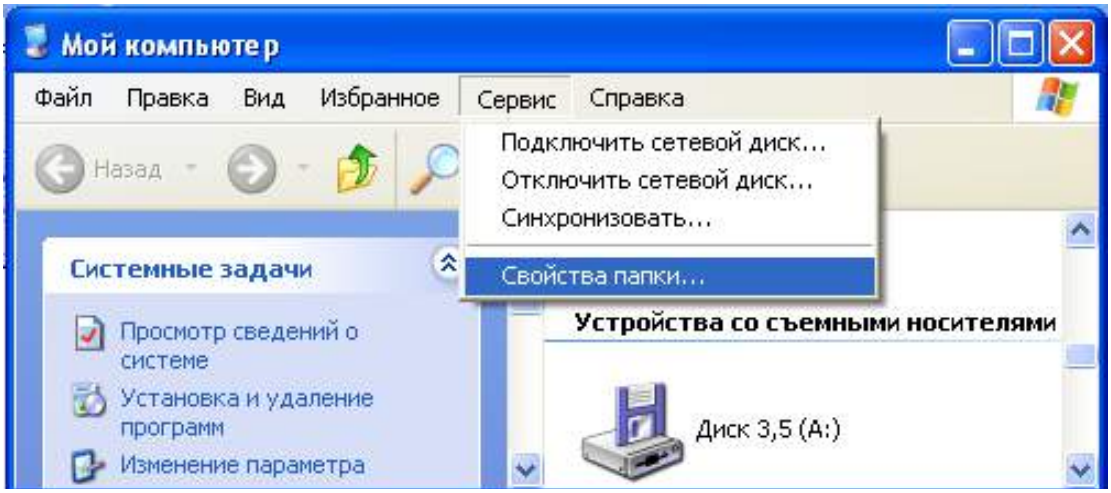
Сохраните этот файл под любым именем и дайте ему расширение .asm. Чтобы запустить эту программу, надо будет этот файл ассемблировать и скомпоновать. Рекомендую все это переложить на плечи компьютера. Создадим bat-файл следующего содержания:

```
cls
if exist %~n1.exe del %~n1.exe
if not exist %~n1.rc goto over1
\masm32\bin\rc /v %~n1.rc
\masm32\bin\cvtres /machine:ix86 %~n1.res
\masm32\bin\ml /c /Cp /Gz /I\masm32\include \
/coff /nologo %~n1.asm
if errorlevel 1 goto TheEnd
\masm32\bin\Link /SUBSYSTEM:WINDOWS /ALIGN:16 \
/MERGE:.data=.text /LIBPATH:\masm32\lib /NOLOGO \
%~n1.obj %~n1.res
if errorlevel 1 goto TheEnd
del %~n1.res
goto TheEnd
:over1
\masm32\bin\ml /c /Cp /Gz /I\masm32\include \
/coff /nologo %~n1.asm
if errorlevel 1 goto TheEnd
\masm32\bin\Link /SUBSYSTEM:WINDOWS /ALIGN:16 \
/MERGE:.data=.text /LIBPATH:\masm32\lib /NOLOGO \
%~n1.obj
:TheEnd
if exist %~n1.obj del %~n1.obj
```

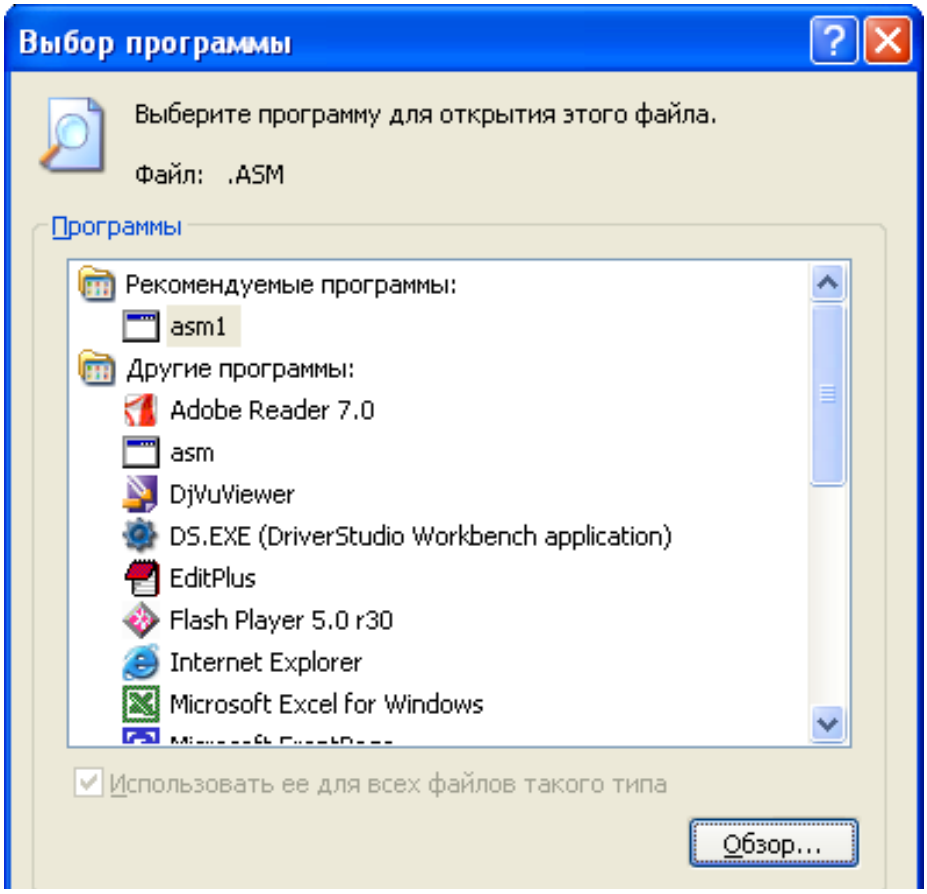
Назовите получившийся bat-файл «asm1.bat» и разместите его в папке Windows/System32. Щелкаем по ярлыку «Мой компьютер».

Мой компьютер→Сервис→Свойства папки→Типы файлов→Создать



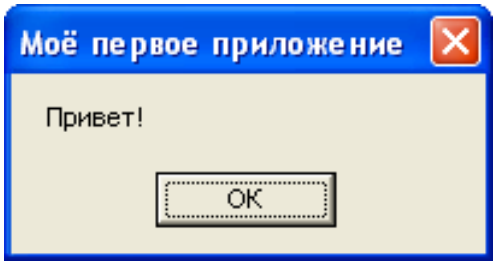



Связываем файлы с расширением .asm с файлом asm1.bat.




Теперь для компиляции и линковки достаточно будет просто щелкнуть по файлу с расширением .asm

Если все набрано правильно, то в текущем каталоге появится файл с именем исходника, но с расширением .exe. Запускаем его и видим результат.



## 4.9. Что при этом происходит?

Ассемблерный код представляет собой мнемоническую версию машинного кода, в котором вместо бинарных кодов операций используются их имена; кроме того, адресам памяти также могут присваиваться имена. Типичная последовательность инструкций выглядит как

```
mov eax,a
add eax,2
mov b,eax
```

Этот код перемещает содержимое памяти по адресу a в регистр eax, затем добавляет к нему константу 2, рассматривая содержимое регистра eax как целое число, и сохраняет результат в ячейке памяти по имени b. Таким образом вычисляется `b := a + 2`

### 4.9.1. Двухпроходный ассемблер

Простейший ассемблер делает два прохода по входному потоку (в данном случае проход – разовое считывание входного файла). При первом проходе находятся все идентификаторы, обозначающие ячейки памяти, и размещаются в таблице символов. Идентификаторам назначаются адреса в памяти, так что после чтения таблица символов может содержать записи, показанные на рисунке. Пусть каждому идентификатору выделено по двойному слову памяти и адреса начинаются с нулевого адреса.

*Таблица 4.9.1*

**Таблица символов ассемблера с идентификаторами a и b**

| Идентификатор | Адрес |
|---|---|
| a | 0 |
| b | 4 |

При втором проходе ассемблер вновь сканирует входной поток. В этот раз он переводит каждый код операции в последовательность битов, представляющих операцию на машинном языке, а каждый идентификатор – в адрес, назначенный идентификатору в таблице символов.

В результате второго прохода получается перемещаемый (relocable) машинный код, что означает, что он может быть загружен в память с любого стартового адреса S. Если S будет добавлено ко всем адресам в коде, то все ссылки будут абсолютно правильны. Поэтому выходной код ассемблера должен различать части инструкций, ссылающиеся на адреса, которые могут быть перенесены.

Машинный код, в который переводятся инструкции:

```
0001 01 00 00000000 *    //mov eax,a
```

```
0011 01 10 00000010      //add eax,2
0010 01 00 00000100 *    //mov b,eax
```

Инструкция представлена в виде двойного слова, в котором первые четыре бита являются кодом инструкции (0001, 0010 и 0011 соответствует загрузке, сохранению и сложению). Под загрузкой и сохранением подразумевается перемещение из памяти в регистр и наоборот. Следующие два бита определяют используемый регистр; 01 означает, что во всех трех командах используется регистр eax. Два последующих бита определяют «дескриптор». 00 означает режим обычной адресации, при котором последующие восемь бит представляют собой адрес памяти; дескриптор 10 указывает на «непосредственный» режим, когда последующие восемь бит являются операндом. Этот режим используется во второй команде. Символ «*» – бит перемещаемости – который имеется у каждого операнда в перемещаемом машинном коде. Предположим, что адресное пространство содержит данные, загруженные начиная с адреса S. В этом случае символ «*» означает, что S должно быть добавлено к адресу операнда. Таким образом, если S=00001111b, то есть 15, то a и b размещаются по адресам 15 и 19. Теперь в абсолютном (или неперемещаемом) машинном коде будет выглядеть как

```
0001 01 00 00001111
0011 01 10 00000010
0010 01 00 00010011
```

У второй команды нет связанного бита перемещаемости, поэтому во второй команде прибавления S не происходит (так как восьмибитовое значение представляет собой не адрес 2, а константу 2).

## Контрольные вопросы

1. Что такое программная привязка?
2. Каков максимальный размер COM-файла?
3. Какие сегменты можно определить в программе, которая будет преобразована в COM-файл?
4. Исходные коды находятся в файле с именем EXAMPLE.ASM. Напишите команды для создания COM-файла с этим же именем.
5. Что конкретно подразумевает директива END: а) завершение сегмента кода, б) завершение программы, в) завершение сегмента данных.
6. Объясните назначение каждого из следующих файлов:

   а) file.ASM, б) file.BAK, в) file.INC, г) file.OBJ, д) file.EXE.

7. Укажите различия в назначениях END и RET.
8. Как разрабатывается программа на языке ассемблера?
9. Назовите основные этапы получения выполняемой программы.
10. Вспомните основные опции транслятора.

11.Для чего нужен отладчик?
12.Можно ли написать программу в машинных кодах?

# ГЛАВА 5
# ОСНОВНЫЕ ПРАВИЛА НАПИСАНИЯ ПРОГРАММ НА ЯЗЫКЕ АССЕМБЛЕРА

Данные правила относятся не только к программированию на языке ассемблера, но и к программированию на других языках. Может быть, их трудно понять, не имея навыка в программировании, но «незнание основ не освобождает от ответственности».

### *Начинайте с комментариев*

Начните с написания инструкции для пользователя – для чего создается и каковы возможности вашей программы. А теперь немного усложните вашу инструкцию по применению вашей программы, подразумевая под «пользователем» программиста, использующего написанный вами код – зачастую этим программистом-исследователем будете вы сами.

Акт записи на обычном языке описания того, что делает программа и что делает каждая функция в программе, является критическим шагом в мыслительном процессе. Хорошо построенное, грамматически правильное предложение – признак ясного мышления. Если вы не можете это записать, то велика вероятность того, что вы не полностью продумали задачу или метод ее решения. Плохая грамматика и построение предложения являются также показателем поверхностного мышления. Поэтому первый шаг в написании любой программы – записать, что именно и как делает программа.

Итак, комментарии для вашей программы уже готовы. Теперь возьмите ваше описание по использованию и добавьте вслед за каждым абзацем блоки кода, реализующие функции, описанные в этом абзаце. Оправдание: «У меня не было времени, чтобы добавить комментарии» на самом деле означает – «Я писал этот код без проекта системы и у меня нет времени воспроизвести его». Если создатель программы не может воспроизвести идеи, воплощенные в программный проект, то кто же тогда сможет?

Работа программиста состоит из двух частей: разработать приложение для пользователя и сделать возможным дальнейшее сопровождение программы. Единственный способ решить вторую часть задачи – комментировать код. Причем комментарии должны описывать не только, что делает код, но и предположения, принятый подход и причины, по которым вы выбрали именно его. Кроме того, необходимо, чтобы комментарии также соответствовали коду. Пишите код так, словно тот, кто будет заниматься его поддержкой, – опасный психопат, знающий где вы живете. Хотя вы можете считать, что ваш код полностью очевиден и может служить примером ясности, без правильных комментариев понять его постороннему достаточно трудно. Парадокс заключается в том, что спустя неделю вы сами можете оказаться в роли этого постороннего.

Старайтесь использовать следующий подход при написании комментариев:

- перед каждой функцией или методом размещается одно или два предложения со следующей информацией:
  - что делает программа;
  - возникающие при этом предположения о программе;
  - что должно содержаться во входных параметрах;
  - что должно содержаться во выходном параметре в случае успешного или неудачного завершения;
  - все возможные выходные значения;
- перед каждой не совсем очевидной частью функции следует поместить одно или два предложения, объясняющие выполняемые действия;
- любой интересный алгоритм заслуживает подробного описания;
- любая нетривиальная ошибка, устраненная в коде, должна комментироваться, при этом нужно привести номер ошибки и описать сделанное исправление;
- правильно размещенные операторы диагностики, проверки условий, а также соглашения об именах переменных могут также служить хорошими комментариями и передавать содержание кода;
- писать комментарии так, будто сами собираетесь заниматься его поддержкой через пять лет;
- если возникла мысль «это хитро сделано» или «это ловкий трюк» – лучше переписать данную функцию, а не комментировать ее.

Если вы будете правильно писать комментарии – вы в безопасности, даже если программист из службы поддержки окажется психопатом.

### *Читайте код*

Все писатели – это читатели. Вы учитесь, когда смотрите, что делают другие писатели. Я настоятельно рекомендую, чтобы, как минимум, члены группы программирования читали код друг у друга. Читатель может найти ошибки, которые вы не увидели, и подать мысль, как улучшить код. Для вас лучше присесть с коллегой и просто разобрать код строка за строкой, объясняя, что и как делается, получить какую-то обратную связь и совет. Для того чтобы подобное упражнение принесло пользу, автор кода не должен делать никаких предварительных пояснений. Читатель должен быть способен понимать код в процессе чтения. Если вам пришлось объяснять что-то вашему читателю, то это значит, что ваше объяснение должно быть в коде в качестве комментария. Добавьте этот комментарий, как только Вы его произнесли; не откладывайте этого до окончания просмотра.

### *Разлагайте сложные проблемы на задачи меньшего размера*

На самом деле это также и правило литературного стиля. Если очень трудно объяснить точку зрения за один раз, то разбейте изложение на меньшие части и по очереди объясняйте каждую. То же самое назначение у глав в книге и параграфов в главе.

### *Используйте язык полностью*

Некоторые программисты считают одним из недостатков языка ассемблера большее, по сравнению с языками высокого уровня, количество команд, но, по-моему, это одно из достоинств языка ассемблера.

### *Проблема должна быть хорошо продумана перед тем, как она сможет быть решена*

Это относится не только к программированию на языке ассемблера.

### *Отредактируйте свой код. Программа должна писаться не менее двух раз*

Раньше, когда вы изучали в школе литературу, вам никогда не приходило в голову сдавать черновик письменного задания, если Вы, конечно, рассчитывали на оценку выше тройки. Тем не менее, многие компьютерные программы являются просто черновиками и содержат столько же ошибок, сколько и черновики ваших сочинений. Хороший код программы должен быть сначала написан, а затем отредактирован в целях улучшения (под «редактированием» имеется в виду «исправление»). Редактирование, как правило, приводит к сокращению кода, а небольшие программы выполняются быстрее.

### *Оптимизация программ на языке ассемблера*

Итак ваша программа заработала, а теперь постарайтесь переделать ее так, чтобы она стала максимально компактной и в тоже время максимально быстродействующей.

Такая оптимизация достигается в три этапа:

- Алгоритмическая оптимизация то есть подбор алгоритма, который выполняет вашу задачу более быстрым способом и позволит сократить не пять, а пятьдесят операторов;
- Подстройка программы под конкретное оборудование;
- Замена некоторых ассемблерных команд на машинный код. Тщательный анализ машинного кода, вырабатываемого транслятором, позволяют прийти к выводу, что некоторые коды человек может выработать более оптимально, чем программа.

ГЛАВА 6

# СИНТАКСИС АССЕМБЛЕРА

## 6.1. Лексемы

*Лексема* – смысловая единица языка ассемблер. К лексемам относятся имена или идентификаторы, числа, константы, зарезервированные слова, операторы и строки.

Каждая единица информации, хранимая в ячейках памяти компьютера, имеет свой адрес. На практике заранее неизвестно, в каких конкретно ячейках памяти во время работы программы будут записаны ее данные, поэтому в языках программирования введено понятие переменной, позволяющее отвлечься от конкретных адресов и обращаться к содержимому памяти с помощью *идентификатора* или *имени*. Имена указывают на значение, о реальном адресе и способе хранения которого можно забыть. В процессе работы содержимое соответствующих ячеек можно менять, обращаясь к переменной по имени. Кроме имени и значения, переменная обычно имеет тип, определяющий, какая информация хранится в данной переменной (число, адрес и т.д.). В зависимости от объема памяти, отведенного для хранения переменной, оно (значение) должно укладываться в допустимый диапазон. Например, значение типа байт имеет диапазон от –128 до 255. Максимальное беззнаковое число равно $2^8 – 1=255$ (8 по количеству битов в байте), минимальное беззнаковое число 0. Максимальное число со знаком равно 127, минимальное число со знаком –128 (1 бит для знака и 7 бит для значения).

### 6.1.1. Идентификаторы

*Идентификаторы* – последовательность из латинских букв, цифр и знаков «?», «.», «@» «_», «$». Ограничение набора символов восходит к ранним языкам программирования, которые возникли еще в эпоху 6-битной кодировки символов.

При записи идентификатора придерживаются следующих правил:

- длина идентификатора, не может быть длиннее 247 символов;
- идентификатор не должен начинаться с цифры;
- точка может быть только первым символом идентификатора (внутри идентификатора точка может использоваться только для разделения полей структуры);
- в идентификаторах большие и малые одноименные буквы при использовании ключа /Ср считаются неэквивалентными;
- в идентификаторах нельзя использовать русские буквы и символы псевдографики;

- знак «$» и «?» имеют самостоятельное значение, поэтому их не рекомендуют к использованию в идентификаторе.

Идентификаторы делятся на *служебные слова* и *имена*. *Служебные слова* имеют заранее определенный смысл, они используются для обозначения таких объектов, как регистры, названия команд и т.д. Все остальные идентификаторы называются именами. *Именами* обозначаются *переменные*, *метки* и другие объекты программы.

### 6.1.2. Целые числа

*Целые числа* могут быть записаны в десятичной, двоичной, восьмеричной и шестнадцатеричной системах счисления. Десятичные числа записываются как обычно, а вот при записи чисел в других системах счисления в конце числа ставится спецификатор – буква, которая указывает, в какой системе счисления записано это число. В конце двоичного числа ставится буква b (binary), в конце восьмеричного – o (octal) или буква q, в конце шестнадцатеричного числа – буква h (hexadecimal), в конце десятичного числа можно поставить букву d (decimal), хотя употребление спецификатора в этом случае и не обязательно.

При записи шестнадцатеричных чисел соблюдают следующие правила:

- если число начинается с «символа» (A-F), то в начале числа ставят ноль;
- хотя при записи числа большие и малые одноименные буквы считаются эквивалентными, но цифры записывают большими буквами, а спецификатор маленькой.

### 6.1.3. Символьные и строковые константы

Для текстовых символов используют либо систему ASCII (*American Standard Code for Information Interchange*), либо Windows-кодировку. Любая *символьная константа* состоит просто из одного символа ASCII. Символьные константы заключаются либо в одинарные, либо в двойные кавычки. Левая и правая кавычки должны быть одинаковы. *Строковые константы* содержат два или более символов ASCII. Строковые константы также заключаются либо в одинарные, либо в двойные кавычки.

- В качестве символьных констант можно использовать русские буквы и символы псевдографики.
- В строковых константах большие и малые одноименные буквы не отождествляются.
- Если в качестве символьных констант или внутри строковых константах надо указать кавычку, то, если символьная или строковая константа заключены в одинарные кавычки, одинарную кавычку надо удваивать, а двойную не надо, и наоборот, если внешние кавычки двойные, то двойная кавычка должна удваиваться, а одинарная не удваивается. Можно также просто

указать ASCII код кавычек: 27h код апострофа «’», 22h код двойных кавычек «”».

*Примеры:*

```
”внутри строки допустим вот такой символ ’ ”
’строки ”могут быть вложены” таким образом’
```

Слова типа can’t или don’t можно записать вот так:

```
1)’can’’t’,2)’can’,27h,’t’,3)”can’t”
```

## 6.2. Предложения

Программа на языке ассемблера – это последовательность предложений, каждое из которых записывается в отдельной строке. Переносить предложение на следующую строку можно используя символ «\», записывать два предложения в одной строке нельзя. Если в предложении более 255 символа, то 256-ой и все следующие за ним символы игнорируются. Правила расстановки пробелов в предложении:

- пробел обязателен между двумя стоящими рядом идентификаторами и/или числами;
- внутри идентификаторов и чисел пробелы не допустимы;
- там где допустим один пробел, можно ставить любое число пробелов.

Эти правила не относятся к пробелам внутри строк, где пробел – обычный значащий символ.

По смыслу все предложения делятся на три группы:

1. комментарии;
2. команды;
3. директивы (приказы ассемблеру).

*Комментарии* предназначены не для микропроцессора, а для людей, они поясняют смысл и детали реализации программы.

*Команды* управляют работой микропроцессора. Для команд транслятор с языка ассемблера всегда генерирует код машинных команд.

*Директивы* управляют работой компилятора по компоновке программы, а не работой микропроцессора. Директивы используются для сообщения компилятору, какие константы и переменные применяются в программе. Как они должны быть расположены в памяти компьютера. Большинство директив не генерирует код машинных команд.

### 6.2.1. Комментарии

Комментарии не влияют на смысл программы, при трансляции ассемблер игнорирует строки комментария. Комментарием считается любая строка, начинающаяся со знака «точка с запятой». Перед знаком «точка с запятой» может быть любое количество пробелов. В комментариях можно использовать русские буквы и символы псевдографики. Предложения-комментарии используются для пояснения не одной команды, а целой группы команд, следующих за этим комментарием. Старайтесь комментировать

задачу, а не инструкции ассемблера. Пустые строки используют для отделения одной части программы от другой – для наглядности. В языке ассемблера допустим и многострочный комментарий. Многострочный комментарий должен начинаться со строчки COMMENT. В качестве маркера многострочного комментария берется первый за словом COMMENT символ, отличный от пробела; этот символ начинает комментарий. Концом многострочного комментария является конец первой из последующих строк программы, в которой в любой позиции снова встретился этот же маркер. Такой вид комментария обычно используется, когда, например, при отладке программы необходимо временно исключить какой-либо фрагмент программы.

### 6.2.2. Команды

Предложения-команды – символьная форма записи машинных команд. Общий синтаксис предложения-команды таков:

[<*метка*>:] <*мнемокод*> [<*операнды*>] [;<*комментарий*>]

#### Метка

*Метка* – это имя. Каждая метка может быть определена только однажды и будет доступна из любой части кода (даже перед местом, где она была определена). Существуют разные способы определения меток. Простейший из них – двоеточие после названия метки. За этой директивой на той же строке даже может следовать другая инструкция. Она определяет метку, значение которой равно смещению точки, в которой она определена. Этот метод обычно используется, чтобы пометить места в коде. Другой способ – это следование за именем метки (без двоеточия) какой-нибудь директивы описания данных. Метке присваивается значение адреса начала определенных в директиве данных и запоминается компилятором как метка для данных с размером ячейки, заданной директивой.

Метка может быть обработана как константа со значением, равным смещению помеченного кода или данных. Например, если вы определяете данные, используя помеченную директиву «char db 224», для того, чтобы поместить адрес начала этих данных в регистр EBX, вам нужно использовать инструкцию «mov ebx,offset char», а для того, чтобы поместить в регистр DL значение байта, на который ссылается «char», нужно использовать «mov dl,char». Последний и самый гибкий способ задания меток – это использование директивы «label». Перед этой директивой должно следовать имя метки, далее, размер оператора, и далее, числовое выражение, определяющее адрес, на который данная метка должна ссылаться. Метка нужна для ссылок на команду из других мест программы, например, для перехода на команду, перед которой стоит метка.

## Мнемокод

*Мнемокод* («легко запоминающийся код») является обязательной частью команды. Это служебное слово, указывающее в символьной форме операцию, которую должна выполнить команда.

## Операнды

*Операнды* команды, если они есть, отделяются друг от друга запятыми. Операнды обычно записываются в виде выражений. Частными случаями выражений являются числа и имена переменных.

Операнды, которые используются в командах ассемблера, могут быть регистром *reg*, адресом памяти *mem*, непосредственным значением, задаваемым прямо в команде *imm*, сегментным регистром *sreg*.

## Машинные коды для всех возможных сочетаний операторов команды

Описание машинного кода производится в шестнадцатеричном виде. При описании машинного кода используются следующие обозначения:

/**цифра** – (цифра от 0 до 7) показывает, что байт mod r/m кода операции использует только операнд r/m. Поле reg содержит цифры которые обеспечивает расширение опткода;

**/r** – показывает, что байт mod r/m команды содержит как регистровый операнд, так и операнд r/m;

**cb, cw, cd, cp, cq** – одно-, двух-, четырех, шести-, восьмибайтное значение, следующее за полем код операции и используемое для смещения сегменте кода и возможно задает новое значение для регистра;

**ib, iw, id, iq** – одно-, двух-, четырех-, восьмибайтное непосредственное значение (число). Следует за опкод, Mod R/M или SIB (если таковые есть), при этом код операции определяет, является ли непосредственный операнд знаковым значением, а все слова, двойные и четверные слова приводятся в порядке «младший байт по младшему адресу»;

**+rb**, **+rw**, **+rd**, **+rq, +i** – код регистра от 0 до 7. Добавляется к байту слева от знака «+». В результате получается окончательный опкод.

*Таблица 6.2.1*

**Коды регистров**

| | | w=0 | w=1 | | | | segment | debug | control | test |
|---|---|---|---|---|---|---|---|---|---|---|
| | bin | rb | rw | rd | rq | i | | | | |
| 0 | 000 | AL | AX | EAX | RAX | ST(0) | ES | DR0 | CR0 | – |
| 1 | 001 | CL | CX | ECX | RCX | ST(1) | CS | DR1 | – | – |
| 2 | 010 | DL | DX | EDX | RDX | ST(2) | SS | DR2 | CR2 | – |
| 3 | 011 | BL | BX | EBX | RBX | ST(3) | DS | DR3 | CR3 | TR3 |
| 4 | 100 | AH | SP | ESP | RSP | ST(4) | FS | – | CR4 | TR4 |
| 5 | 101 | CH | BP | EBP | RBP | ST(5) | GS | – | – | TR5 |
| 6 | 110 | DH | SI | ESI | RSI | ST(6) | – | DR6 | – | TR6 |
| 7 | 111 | BH | DI | EDI | RDI | ST(7) | – | DR7 | – | TR7 |

Машинные коды команды сопровождаются описанием синтаксиса команды для соответствующего сочетания операндов. При описании синтаксиса используются следующие обозначения:

**rel8** – относительный адрес (смещение). Задаёт смещение от 128 байт перед концом инструкции до 127 байт после конца инструкции. Иными словами, смещение от -128 до 127 байт относительно адреса после конца инструкции;

**rel16, rel32, rel64** – относительные адреса в пределах сегмента кода, содержащего данную команду, используемые для операндов с размером операнда 16 (use16), 32 (use32) и 64 (use64) бита соответственно;

**ptr16:16, ptr16:32, ptr 16:64** – непосредственное значение 20-, 48- и 80-разрядного адреса памяти, задаваемое прямо в команде, дальний указатель (обычно на адрес в сегменте кода, отличном от текущего), используется при установленном атрибуте размера операнда 16, 32 или 64 бита (первая часть указателя является селектором или значением сегментного регистра кода, вторая – смещением в целевом сегменте кода);

**reg** – операнд в одном из регистров размером в байт, слово, двойное слово, четверное слово;

**r8** – операнд в одном из регистров размером в байт: AH, AL, BH, BL, CH, CL, DH, DL, BPL, SPL, DIL, SIL или один байт регистров (R8L – R15L) доступный, когда используется REX.R и 64-битный режим;

**r16** – операнд в одном из регистров размером в слово: AX, BX, CX, DX, BP, SP, SI, DI или одно слово регистров (R8 – R15) доступный, когда используется REX.R и 64-битный режим;

**r32** – операнд в одном из регистров размером в двойное слово: EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI или одно двойное слово регистров (R8D – R15D) доступный, когда используется REX.R и 64-битный режим;

**r64** – операнд в одном из регистров размером в четверное слово RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8 – R15 доступные, когда используется REX.R и 64-битный режим;

**imm, imm8, imm16, imm32, imm64** – непосредственный одно-, двух-, четырех-, восьмибайтовый операнд команды;

**mem, m8, m16, m32, m48, m64, m128** – операнд в памяти размером в байт, слово, двойное слово, 48/64/128 бит;

**m8** – адрес однобайтной ячейки памяти в регистре (E)SI или (E)DI. Используется только со строковыми инструкциями;

**m16** – адрес двухбайтной ячейки памяти в регистре (E)SI или (E)DI. Используется только со строковыми инструкциями;

**m32** – адрес четырёхбайтной ячейки памяти в регистре (E)SI или (E)DI. Используется только со строковыми инструкциями;

**m64** – адрес 64-битной (учетверённое слово) ячейки памяти. Используется только с инструкцией CMPXCHG8B;

**m128** – адрес 128-битной (увосьмерённое слово) ячейки памяти. Используется только с инструкциями SSE и SSE2;

**r/m8** – байтовый операнд, который содержится либо в одном из регистров размером один байт, либо в ячейке памяти размером один байт;

**r/m16** – операнд в регистре размером в слово или в ячейке памяти размером в слово (используется в командах, для которых размер операнда равен 16 бит);

**r/m32** – операнд в регистре размером в двойное слово или в ячейке памяти размером в двойное слово (используется в командах, для которых размер операнда равен 32 битам);

**m16:16, m16:32, m16:64** – операнд в памяти, содержащий дальний указатель, находящийся по данному адресу;

**m16&32**, **m16&16**, **m32&32** – адрес ячейки памяти, состоящей из нескольких элементов, чьи размеры задаются справа и слева от амперсанда. Доступны все режимы адресации памяти. Операнды m16&16 и m32&32 используются инструкцией BOUND (в них задаются верхние и нижние границы массива). Операнд m16&32 используется LIDT и LGDT.

**moffs8, moffs16, moffs32, moffs64** – переменная (смещение в памяти), типа байт, слово, двойное, четверное слово, требующая для выполнения некоторых вариантов команды mov (байт mod r/m не используется, адрес задается простым смещением относительно базы сегмента). Используется некоторыми вариантами инструкции MOV;

**sreg** – сегментный регистр. Номера сегментных регистров в поле Reg байта ModR/M: ES=0, CS=1, SS=2, DS=3, FS=4, GS=5;

**port8** – 8-разрядный адрес в пространстве ввода/вывода;

**m32fp, m64fp, m80fp** – адрес в памяти операнда с плавающей точкой одинарной точности, двойной точности или расширенной точности. Используется инструкциями FPU;

**m16int, m32int, m64int, m80int** – целочисленный операнд в памяти, используемый в командах сопроцессора;

**ST** или **ST(0)** – верхний элемент стека сопроцессора;

**ST(i)** – *i*-ый элемент стека сопроцессора (*i*=0…7);

**rmmx0…rmmx7** – операнд в одном из регистров целочисленного расширения MMX;

**rmmx/m32** – младшая часть (32 бита) MMX-регистра или 32-разрядный операнд в памяти;

**rmmx/m64** – MMX-регистр или 64-разрядный операнд в памяти;

**rxmm0…rxmm7** – операнд в одном из 128-битный XMM-регистров расширения MMX с плавающей точкой;

**rxmm/m32** – XMM-регистр (от XMM0 до XMM7) или адрес 32-битной ячейки памяти;

**rxmm/m64** – XMM-регистр (от XMM0 до XMM7) или адрес 64-битной ячейки памяти;

**rxmm/m128** – XMM-регистр (от XMM0 до XMM7) или адрес 128-битной ячейки памяти.

В таблице указывается, какие типы процессоров поддерживают данную мнемонику:

| Сокращенное наименование | Полное наименование |
|---|---|
| | **Intel** |
| P | Pentium |
| PII | Pentium II (MMX) |
| SSE | SSE (Katmai NI) |
| SSE2 | SSE2 |
| SSE3 | SSE3 (Prescott NI) |
| E64T | 64-bit Memory |
| | **AMD** |
| K6 | K6 |
| 3D! | 3DNow! |
| 3Mx+ | 3DNow! and MMX Ext. |
| A64 | AMD64 |

### Комментарий

В конце команды можно поместить комментарий, для чего надо поставить точку с запятой и выписать любой текст, который будет рассматриваться как комментарий. Такой комментарий, в отличие от комментариев-предложений, обычно используется для пояснений именно данной команды.

## 6.2.3. Директивы

Переменные с указанием их типа вводятся в программу с помощью специальных команд описания – *директив*. Это позволяет компилятору организовать эффективное хранение и обработку данных и повышает ясность исходных текстов. Каждый тип описывается своим ключевым словом.

Директивы служат для сообщения о том, какие константы и переменные используются в программе.

Основное отличие команды от директивы:

*команда* языка ассемблер *всегда* генерирует машинный код и предназначена для управления *микропроцессора*;

*директива* (псевдооператор, псевдокоманда) управляет работой *компилятора* и *не генерирует* машинный код.

Основное различие между диалектами языка ассемблер (tasm, masm, fasm и т.д.) – это различие в директивах.

*Синтаксис директив:* [<*имя*>] < *название директивы*><*операнд*>

Имя, указываемое в начале директивы – это имя константы или переменной, описываемой данной директивой. Название директив – это служебное слово.

### Директивы определения данных

*Директивы определения данных* (или *директивы резервирования и инициализации данных*) необходимы для описания типов переменных (байт, слово, двойное слово и т. д.), с которыми работает программа. Директивы определения данных являются указаниями транслятору на выделение определённого объёма памяти. Директива начинается с D (**D**efine, *определить*), после которой идет сокращение от размера определяемого элемента данных (DB – **B**yte *байт*, DW – **W**ord *слово*, DD – **D**ouble word *двойное слово*, DF – **F**ar pointer word *указатель на дальнее слово*, DP – **P**ointer *указатель*, DQ – **Q**uadword *учетверенное слово*, DT – **T**en bytes *10 байт*).

DF – *длинный сегментированный указатель* (32 бита смещения + 16 бит сегмента), синоним: DP.

Кроме целых чисел, как аргументы, можно указывать *вещественные*. DD может также использоваться для описания и хранения *коротких* (single-precision) *вещественных* чисел $+/-1{,}38\times10^{-38}$ …$3{,}40\times10^{38}$, DQ для *длинных* (long, double) *вещественных* $+/-2{,}33\times10^{-308}$... $1{,}79\times10^{308}$, а DT для *временных* (temporary) *вещественных* $+/-3{,}37\times10^{-4932}$... $1{,}18\times10^{4932}$ (табл. 6.2.2.).

Помимо интерпретации значений в соответствующих ячейках как целых, они могут интерпретироваться следующим образом: DW – int/unsigned или 16-битное смещение; DD – long, float, 16-битный far (сегмент:смещение) или 32-битный близкий (смещение) указатель; DF – 32-битный far (дальний) указатель типа сегмент:смещение; DQ – double; DT – упакованное десятичное (packed BCD) число длинной 20 цифр или long double (оба – форматы представления числа для сопроцессора).

Все выше перечисленные директивы можно использовать для строкового значения (представление чисел в ввиде ASCII кодов).

String_1 DB *'a'*
String_2 DW *'ab'*
String_3 DD *'abcd'*
String_4 DF *'abcdef'*
String_5 DQ *'abcdefgh'*
String_6 DT *'abcdefghij'* ;
String_7 DB '*я программирую на ассемблере!*' Директива DB резервирует и инициализирует столько байт, сколько символов нам вздумалось написать между кавычками или апострофами.

*Таблица 6.2.3*

**Директивы определения данных**

| **Директива и ее синонимы** | **Размер в байтах** | **Диапазон принимаемых значений аргументов** | | | |
|---|---|---|---|---|---|
| | | **в шестнадцатеричной системе счисления** | **в десятеричной системе** | | |
| | | | **беззнаковые целые числа** | **знаковые целые числа** | **вещественные числа** |
| DB/ BYTE/ SBYTE | 1 | От 0 до 0FFh | От 0 до 255 | От –128 до +127 | – |
| DW/ WORD/ SWORD | 2 | От 0 до 0FFFFh | От 0 до 65535 | От $-2^{15}$ до $+2^{15}-1$ | – |
| DD/ DWORD/ SDWORD/ REAL4 | 4 | От 0 до 0FFFFFFFFh | От 0 до $2^{32}-1$ | От $-2^{31}$ до $+2^{31}-1$ | От +/– $1{,}18\times10^{-38}$ до $3{,}4\times10^{38}$ |
| DF/ DP/ FWORD | 6 | От 0 до 0FFFFFFFFFFFFh | От 0 до $2^{48}-1$ | От $-2^{47}$ до $2^{47}-1$ | – |
| DQ/ QWORD/ REAL8 | 8 | От 0 до 0FFFFFFFFFFFFFFFFh | От 0 до $2^{64}-1$ | От $-2^{63}$ до $2^{63}-1$ | От +/– $2{,}23\times10^{-308}$ до $1{,}79\times10^{308}$ |
| DT/ TBYTE/ REAL10 | 10 | От 0 до 0FFFFFFFFFFFFFFFFFFFFh | От 0 до $2^{80}-1$ | От $-2^{79}$ до $2^{79}-1$ | От +/– $3{,}37\times10^{-4932}$ до $1{,}18\times10^{4932}$ |

*Таблица 6.2.3*

**Интерпретация директив DW, DD, DP и DQ как адресные выражения**

| Директива | Адресное выражение |
|---|---|
| DW | 16-битный адрес сегмента; смещение в 16-битном сегменте |
| DD | 16-битный адрес + 16-битное смещение; 32-битный адрес |
| DF | 16-битный адрес сегмента + 32-битное смещение |
| DQ | 16-битный адрес сегмента + 64-битное смещение |

### *Директива DB*

По директиве DB (**D**efine **B**yte, *определить байт*) определяются данные размером в байт.

*Синтаксис директивы*:

[<*имя*>] DB <*операнд*> {,<*операнд*>} [;<*комментарий*>]

*Синонимы*: BYTE (от 0 до 255), SBYTE (от –128 до +127)

Встретив такую директиву, ассемблер вычисляет операнды и записывает их значения в последовательные байты памяти. Первому из этих байтов дается указанное имя, по которому на этот байт можно ссылаться из других мест программы.

Существуют следующие способы задания операндов директивы DB:

1) ? (знак неопределенного значения);
2) константное выражение со значением от –128 до 255;
3) символьная строка из одного или более символов, заключенная в кавычки.

#### *Операнд «?»*

*Пример*:

```
X DB ?
```

По этой директиве описывается переменная X. Для нее отводится один байт в памяти, в который ничего не записывается. В этом случае говорят, что переменная не получила никакого значения. Выделив байт под переменную, ассемблер запоминает ее адрес. Когда ассемблер снова встретит в тексте программы имя этой переменной, то он заменит имя на данный адрес.

Адрес ячейки, выделенной переменной с именем X, принято называть значением имени X. По описанию переменной ассемблер запоминает, сколько байт занимает переменная в памяти. Этот размер называется типом переменной. Значение (адрес) и тип (размер) имени переменной однозначно определяют ячейку памяти, обозначаемую этим именем.

#### *Операнд – константное выражение со значением от –128 до 255*

Применяется для описания переменной размером в байт с начальным значением в виде числа величиной от –128 до 255. Один байт представляет либо знаковое число в пределах от –128 до +127, либо беззнаковое число

от 0 до 255. Максимальное число 255 равно $2^8 - 1$ (8 по количеству битов в байте).

```
A  DB    254         ;0FEh
B  DB    -2          ;0FEh (256-2=254)
C  DB    0FEh        ;0FEh
D  DB    11111110b   ;0FEh
```

По каждой из этих директив ассемблер отводит один байт под переменную и записывает в этот байт число. К началу выполнения программы переменная A будет иметь значение 254, переменная B – значение – 2, переменная C – значение 0FEh, а переменная D 11111110b. В качестве значения переменной может быть указан символ. В качестве символа можно указать как его код (в кодировке ASCII) либо указать сам символ в кавычках.

### *Директива с несколькими операндами*

Для описания переменной-массива с некоторыми начальными значениями, применяется директива DB с несколькими операндами.

*Пример*:
```
M       DB      2
        DB      -2
        DB      ?
        DB      '*'
```

В массивах имя дается только его первому элементу, а остальные остаются безымянными. Если в массиве много элементов, то такой способ описания массива слишком громоздок. Поэтому допускается также и упрощенная форма записи: `M  DB  2,-2,?,'*'`

По директиве DB с несколькими операндами ассемблер выделяет в памяти соседние байты памяти по одному на каждый операнд, и записывает в эти байты значения операндов (для операнда «?» ничего не записывается).

### *Операнд – строка*

Если в директиве несколько соседних операндов символы, то их можно объединить в одну строку. Два следующих примера эквивалентны:

```
S   DB  'к','о','т'        S   DB  'кот'
```

Вопрос о том, объединять соседние символы в одну строку или нет, а если объединять, то какие именно, решает сам автор программы. Правильной будет и такая запись:

```
S   DB  'ко','т'
```

и такая

```
S   DB  'к','от'
```

В любом случае каждая из этих директив является эквивалентом следующей директивы:

```
S   DB   'к'
    DB   'о'
    DB   'т'
```

***Операнд – конструкция повторения DUP***

Довольно часто в директиве приходится указывать одинаковые операнды. Например, при описании байтового массива R из 8 элементов, где каждый элемент проинициализирован 0, можно записать так:

```
R           DB      0, 0, 0, 0, 0, 0, 0, 0
```

А можно записать короче:

```
R           DB      8 DUP (0)
```

В этой директиве в качестве операнда использована конструкция повторения, в которой сначала указывается коэффициент повторения, затем служебное слово DUP (**DUP**LICATE *копировать*), а за ним в круглых скобках – повторяемая величина.

В общем случае эта конструкция имеет следующий вид:

$$k \quad \text{DUP} \ (p_1, p_2, \ldots, p_n),$$

где $k$ – константное выражение с положительным значением, n≥1, $p_i$ – любой допустимый операнд директивы DB (в частности, это может быть снова конструкция повторения):

$$\underbrace{p_1, p_2, \ldots, p_n, p_1, p_2, \ldots, p_n \ \ldots, p_1, p_2, \ldots, p_n}_{k \text{ раз}}$$

***Рис. 6.2.1***

Например, директивы слева эквивалентны директивам справа:

```
X  DB  2 DUP ('ab', ?, 1)  | X  DB  'ab',?,1,'ab',?,1
Y  DB  -7,3DUP(0,2DUP(?))  | Y  DB  -7,0,?,?,0,?,?,0,?,?
```

Вложенность конструкций DUP используют для описания многомерных массивов. Директива

```
A    DB   200 DUP (300 DUP (?))
```

отводит в памяти место под байтовую матрицу $A$ размера 200×300, в которой элементы расположены в памяти следующим образом: первые 300 байтов – это элементы первой строки матрицы, следующие 300 байтов – элементы второй строки и т.д. (А слабо вам написать 60 тысяч раз директиву ‘DB ?’?)

### *Директива DW*

Директивой DW (**D**efine **W**ord, *определить слово*) описываются переменные размером в слово. Аналогична директиве DB, вкратце рассмотрим допустимые виды ее операндов.

*Синонимы*: WORD (0 до 65535), SWORD (от $-2^{15}$ до $+2^{15}-1$)

***Операнд «?»***

*Пример:*

```
X DW ?
```

По этой директиве описывается переменная X. Для нее отводится одно слово в памяти, в которое ничего не записывается, т.е. эта переменная не получает начального значения.

***Константное выражение со значением от –32768 до 65535***

Применяется для описания переменной размером в слово с начальным значением в виде числа величиной от –32768 до 65535. Слово представляет либо знаковое число в пределах от –32768 до +32767, либо беззнаковое число от 0 до 65535. Максимальное число 65535 равно $2^{16} - 1$ (16 по количеству битов составляющих слово):

```
A  DW  1234h
B  DW  -2       ;0FFFEh(65536-2=65534)
```

По каждой из этих директив ассемблер отводит одно слово под переменную и записывает в это слово указанное число, которое становится начальным значением этой переменной. Как и в случае директивы DB, неотрицательные числа записываются в память как числа без знака, а отрицательные числа в дополнительном коде. Поэтому числа, которые могут быть заданы как операнды директивы DW, должны принадлежать отрезку $[-2^{15}, 2^{16} - 1]$.

В памяти компьютера числа размером в слово хранятся в «перевернутом» виде, поэтому по нашим двум директивам память заполнится следующим образом:

| 34 | 12 | FE | FF |
|---|---|---|---|
| A | | B | |

***Рис.6.2.2. Представление в памяти директивы DW 1234h,-2***

Частный случай директивы DW строка из одного или двух символов, например:

```
S1   DW    '01'
S2   DW    '1'
```

Если указана строка из двух символов, то ассемблер берет коды указанных символов (код ASCII для ‘0’ равен 30h, для ‘1’ – 31h) и образует

из них число-слово (3031h), которое и считается начальным значением описываемой переменной S1. Но, как и любое число размером в слово, данное значение будет записано в память в «перевернутом» виде. Если же в правой части директивы DW указан один символ, то к нему слева ассемблер приписывает символ с кодом 0 и далее работает с этим символом как с двухсимвольной строкой. Поэтому по нашим двум директивам память будет заполнена следующим образом:

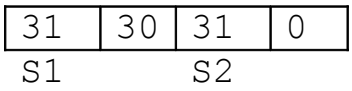



***Рис. 6.2.3. Представление в памяти директивы DW '01','1'***

В связи с тем, что операторы-строки записываются в память в «перевернутом» виде, что, в общем-то, не характерно для строк, то подобные операнды редко указываются в директиве DW, хотя этот способ можно использовать, чтобы скрыть строку от начинающего хакера-взломщика. Первые 128 символов Unicode с кодами от 0000h до 007Fh (цифры, знаки препинания, пробел, символы от ‘A’ до ‘Z’ и от ‘a’ до ‘z’ и т.п.) совпадают с символами ASCII, поэтому для преобразования ASCII символов в UNICODE символы используется следующая формула:

16-битный номер символа в UNICODE =
=(с 15-го по 8-ой биты нули) +(номер символа в ASCII)

Тогда через директиву dw можно передавать латиницу в формате UNICODE следующим образом:

dw ‘H’,‘e’,‘l’,‘l’,‘o’,‘,’,‘ ’,‘w’,‘o’,‘r’,‘l’,‘d’,‘!’,0

Варианты определения слова со значением 256 (100h) и двойного слова со значением 65539 (10003h).

```
dw 256          | dd 65539
dw 100h         | dd 10003h
db 0,1          | dw 3, 1
                | db 3, 0, 1, 0
```

### *Адресное выражение*

В качестве операнда директивы DW может быть указано адресное выражение, т.е. выражение, значением которого является адрес. Основной случай адресного выражения – это имя переменной или метки, например:

```
C    DB    ?
D    DW    C
```

В этом случае ассемблер запишет в слово, выделенное под переменную D, адрес переменной C, который становится начальным значением переменной D.

***Несколько операндов, конструкция повторения***

В правой части директивы DW можно указать любое число операндов, а также конструкцию повторения, например:

```
E   DW    40000, 3 DUP (?)
```

### *Директива DD*

Директивой DD (**D**efine **D**ouble word, *определить двойное слово*) описываются переменные, под которые отводятся двойные слова. В остальном эта директива похожа на две предыдущие.

*Синонимы*: DWORD (0 до 65535), SDWORD (от $-2^{31}$ до $+2^{31}-1$), REAL4 (от +/– $1{,}18\times10^{-38}$ до $3{,}4\times10^{38}$)

***Операнд «?»***

*Пример:*

```
X DD ?
```

По этой директиве описывается переменная X. Для нее отводится двойное слово в памяти, в которое ничего не записывается, т.е. эта переменная не получает начального значения.

***Константное выражение со значением от $-2^{31}$ до $2^{32}-1$***

Применяется для описания переменной размером в двойное слово с начальным значением в виде числа величиной от –2147483648 до 4294967295. Двойное слово представляет либо знаковое число в пределах от –2147483648 до +2147483647, либо беззнаковое число от 0 до 4294967295. Максимальное число равно $2^{32} - 1$ (32 по количеству битов составляющих двойное слово):

```
A   DD   12345678h
```

***Адресное выражение***

Операнд задает абсолютный 32-разрядный адрес.

***Несколько операндов, конструкция повторения***

В правой части директивы DD можно указать любое число операндов, а также конструкцию повторения, например:

```
E  DD  33 DUP (?), 12345678h
```

### *Дополнительные директивы определения данных*

- директива DF и резервирует в памяти 6 последовательно расположенных байтов. *Синонимы директивы*: DP, FWORD;

- директива DQ резервирует в памяти 8 байтов. *Синонимы директивы*: QWORD, REAL8;
- директива DT резервирует в памяти 10 байтов. *Синонимы директивы*: TBYTE, REAL10

Вещественные числа могут использоваться только с директивами DD, DQ и DT. Для записи вещественного числа используется любая комбинация десятичных чисел. Цифры, которые стоят перед десятичной точкой, представляют целую часть, стоящие после точки – дробную часть числа. Цифры, стоящие после экспоненциального символа *E*, представляют степень. Если степень задана, то знак ее показателя определяется стоящими перед ней знаками + или –.

*Примеры:*

```
DD 25.23 ;кодируется как 41C9D70Ah
DD 2.523E1;кодируется как 41C9D70Ah
DD 2523.0E–2;кодируется как 41C9D70Ah
```

Здесь все понятно, так как это одно и тоже число. А теперь напишем одно и тоже число 96.875, но с директивами DD, DQ и DT. Компилятор выдаст нам соответственно: 42C1C0000h, 4058380000000000h, 4005C1C0000000000000h. Попробуйте разобраться самостоятельно, используя материалы главы «Представление данных», почему число:

$$96{,}875 = 96 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 1100000{,}111\text{b} = 1{,}100000111\text{b} \times 2^6$$

было по-разному закодировано компилятором.

## Выводы

Директивы DB, DW, DD, DF, DQ и DT резервируют соответственно байты, двух-, четырех-, шести-, восьми-, десятибайтные слова. После директивы DB (DW, DD, DF, DQ или DT) записывается значение элемента данных, знаком вопроса обозначается неопределенная (произвольная) величина. Следующие друг за другом директивы DB (DW, DD, DF, DQ или DT) можно записать в одну строку, разделив значения запятыми. Обозначения ASCII-символов в директивах DB (DW, DD, DF, DQ или DT) можно сгруппировать, задав их одной строкой в кавычках. Повторяющиеся элементы после директивы DB (DW, DD, DF, DQ или DT) группируются при помощи конструкции DUP. Конструкции DUP допускают вложенность.

## Директивы эквивалентности и присваивания

### *Директива эквивалентности*

Допустим, Вы пишете на языке ассемблера компьютерную игрушку. Через какое-то время Вам понадобилось изменить правила игры и повысить

или понизить порог очков (100), после превышения которого партия считается оконченной. Число 100 упоминается несколько раз в разных местах Вашего файла, и постоянно менять все операторы довольно хлопотно. Замену нельзя выполнить автоматической командой текстового редактора «найти и заменить», так как число 100 может встречаться в разных местах программы и обозначать не только предел набранных очков. Поэтому лучше всего описать число 100 как константу и в дальнейшем обращаться к ней по имени. В языке ассемблера используют константы, которые подобно переменным имеют имена. Единственное их отличие от переменных заключается в том, что им нельзя присвоить новое значение во время выполнения программы. Имя константы может быть более выразительным и в ряде случаев более коротким, чем ее значение. При многократном использовании такой константы в программе экономится время набора текста программы. Изменение значения именованной константы связано с изменением только одного оператора, тогда как в случае использования числовых значений этой константы такое исправление пришлось бы делать многократно.

Константы в языке ассемблера описывают с помощью директивы EQU (**EQU**AL, *равно*).

*Синтаксис директивы:* <*имя*> EQU <*операнд*>

Обязательно должны быть указаны имя и операнд, причем только один. Директивой EQU автор программы заявляет, что указанному операнду он дает указанное имя и требует, чтобы все вхождения данного имени в текст программы ассемблер заменял на этот операнд. Например, если есть директива *STAR EQU '*',* то ассемблер будет рассматривать предложение *N DB STAR* как предложение *N DB '*',* другими словами, *STAR* и '*' – это одно и то же. Директива EQU носит чисто информационный характер и по нему ассемблер ничего не записывает в конечную программу. Поэтому директиву EQU можно ставить в любое место программы.

***Операнд – имя***

Если в правой части директивы указано имя регистра, переменной, константы и тому подобное, тогда имя слева объявляется синонимом данного имени и все последующие вхождения в текст программы этого имени-синонима ассемблер будет заменять на имя, указанное справа.

*Пример:*

```
A   DW   ?
B   EQU  A
C   DW   B  ; эквивалентно C DW A
```

Имена-синонимы обычно используются для введения более удобных и наглядных обозначений.

***Операнд – константное выражение***

*Пример*:

```
N   EQU  100
K   EQU  2*N-1;K=199
P   EQU  'A'
```

Если в правой части директивы EQU стоит константное выражение, тогда указанное слева имя принято называть именем константы. Значением этой константы объявляется значение выражения. N – это константа со значением 100, K – со значением 199, P – со значением 41h (это код буквы ‘A’ в системе ASCII). Все последующие вхождения в текст программы имени константы ассемблер будет заменять на значение этой константы. Например, директива

```
X DB N DUP(?)
```

эквивалентна директиве

```
X DB 100 DUP(?)
```

Случаи, когда полезно применение констант, такие же, как в языках высокого уровня. Например, в качестве констант рекомендуется описывать размеры массивов, поскольку в таком случае легко настроить программу на работу с массивом любого другого размера – для этого достаточно внести изменение лишь в директиву EQU, описывающую константу. Если в константном выражении используются имена других констант, то они (эти константы) должны быть описаны раньше данной директивы EQU, иначе ассемблер, просматривающий текст программы сверху вниз, не сможет вычислить значение этого выражения.

***Операнд – любой другой текст***

*Пример:*

```
S    EQU   'Вы ошиблись'
LX   EQU   X+(N-1)
WP   EQU   WORD PTR
```

В данном случае считается, что указанное имя обозначает операнд в том виде, как он записан (операнд не вычисляется). Именно на этот текст и будет заменяться каждое вхождение данного имени в программе. Например, следующие предложения будут эквивалентны предложениям справа:

```
ANS DB S,'!'  ANS DB 'Вы ошиблись!'
NEG LX        NEG X+(N-1)
INC WP [BX]   INC WORD PTR [BX]
```

Такой вариант директивы EQU обычно используется для того, чтобы ввести более короткие обозначения для часто встречающихся длинных текстов. Текст, указанный в правой части директивы EQU, должен быть сбалансирован по скобкам и кавычкам и не должен содержать вне скобок и кавычек символа «;». Поскольку текст не вычисляется, то в нем можно ис-

пользовать как имена, описанные до этой директивы EQU, так и имена, описанные после нее.

### *Директива присваивания «=»*

*Синтаксис директивы:* *<имя> = <константное выражение>*

Эта директива определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. В отличие от констант, определенных по директиве EQU, данная константа может менять свое значение, обозначая в разных частях текста программы разные числа.

*Пример:*

```
K=10
A           DW   K    ;эквивалентно A DW 10
K=K+4
B           DW   K    ;эквивалентно B DW 14
```

Подобного рода константы используют ради «экономии имен»: если в разных частях программы используются разные константы и области использования этих констант не пересекаются, тогда, чтобы не придумывать новые имена, этим константам можно дать одно и тоже имя.

### *Целочисленные выражения*

Операнды являются элементарными компонентами, из которых формируется часть машинной команды, обозначающая объекты, над которыми выполняется операция. В более общем случае операнды могут входить в более сложные образования, называемые выражениями. Целочисленные выражения представляют собой комбинации целочисленных значений и операторов, рассматриваемые как единое целое. Результатом целочисленных вычислений выражения может быть адрес некоторой ячейки памяти или некоторое константное (абсолютное) значение. В процессе вычисления выражения получается 32-битное число, в диапазоне от 0 до 0FFFFFFFFh.

- Арифметические операторы с учетом порядка их выполнения – от старшего к младшему: скобки «(» и «)»; одноместные (унарные) «+» и «–» (знак числа); оператор умножения «*», целочисленного деления «/», получения остатка от деления «MOD»; двухместные (бинарные) «+» и «–» (операторы сложения и вычитания).

*Пример:*

```
TAB_SIZE EQU 50          ;размер массива в байтах
SIZE_EL EQU 2            ;размер элементов
;вычисляется число элементов массива и заносится в CX
        MOV CX,TAB_SIZE/SIZE_EL     ;оператор «/»
```

Порядок выполнения операторов учитывается в сложных выражениях, состоящих из нескольких арифметических операторов.

```
4 + 5 * 2        ;Сначала умножение, затем сложение
12 – 1 MOD 5     ;Сначала остаток от деления, затем вычитание
-5 + 2           ;Сначала унарный минус, затем сложение
(4 + 2)* 6       ;Сначала сложение в скобках, затем умножение
```

*Таблица 6.2.4*

| Выражение | Значение, которое подставит компилятор |
|---|---|
| 16/5 | 3 |
| −(3+4)*(6−1) | −35 |
| −3+4*6−1 | 20 |
| 25 mod 3 | 1 |

- Операторы сдвига SHL, SHR выполняют сдвиг выражения на указанное количество разрядов. Если указанное число отрицательное, то SHL будет заменен на SHR, а сдвиг произведен на модуль числа.

*Пример:*

```
MASK_B EQU 10111011B
MOV AL,MASK_B SHR 3       ;AL=00010111B
```

- Операторы сравнения EQ, NE, LT, LE, GT, GE (возвращают значение «истина» или «ложь») предназначены для формирования логических выражений.

*Пример:*

```
TAB_SIZE EQU 30           ;размер таблицы
MOV AL,TAB_SIZE GE 50     ;сравнение размера таблицы с 50
;и загрузка в AL в случае меньшего значения 0
CMP AL,0                  ;если TAB_SIZE < 50, то
JE M1                     ;переход на M1
```

- Логические операторы NOT AND OR, XOR выполняют над выражениями побитовые операции.

*Пример:*

```
FLAGS EQU 10010011B
MOV AL,FLAGS XOR 01B      ;AL=1001001 0 b; пересылка в AL
;поля FLAGS с инвертированным правым битом
```

- Индексный оператор [ ]. Транслятор воспринимает квадратные скобки как указание сложить значение перед скобками с выражением, находящимся внутри скобок.

*Пример:*

```
MOV EAX,MAS[ESI]  ;пересылка слова по адресу MAS+[ESI] в регистр EAX
```

- Оператор переопределения типа PTR применяется для временного переопределения или уточнения типа метки или переменной, определяемой выражением. Тип может принимать одно из следующих значений: BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR.
- Оператор переопределения сегмента «**:**» (двоеточие) заставляет вычислять физический адрес относительно конкретно задаваемой

сегментной составляющей: «имя сегментного регистра», «имя сегмента» из соответствующей директивы SEGMENT или «имя группы».

- Оператор именования типа структуры «.» (точка).
- Оператор получения сегментной составляющей адреса выражения SEG возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.
- Оператор получения смещения выражения OFFSET позволяет получить значение смещения выражения в байтах относительно того сегмента, в котором выражение определено.

Выполнение операторов ассемблера при вычислении выражений осуществляется в соответствии с их приоритетами. Операции с одинаковыми приоритетами выполняются последовательно слева направо. Изменение порядка выполнения возможно путем расстановки круглых скобок, которые имеют наивысший приоритет.

В таблице указаны операторы ассемблера в порядке убывания старшинства.

*Таблица 6.2.5*

**Операторы ассемблера в порядке убывания старшинства**

| | **Операторы с равным старшинством** |
|---|---|
| 1 | (), [], <>, LENGTH, SIZE, WIDTH, MASK |
| 2 | . |
| 3 | : |
| 4 | PTR, OFFSET, SEG, TYPE, THIS |
| 5 | HIGH, LOW |
| 6 | Одноместные (унарные) + и - |
| 7 | *, /, MOD, SHL, SHR |
| 8 | Двухместные (бинарные) + и - |
| 9 | EQ, NE, LT, LE, GT, GE |
| 10 | NOT |
| 11 | AND |
| 12 | OR, XOR |
| 13 | SHORT, TYPE |

### *Константные выражения*

Значениями константных выражений всегда являются 32-битовые целые числа (исключениями являются директивы DQ, DF, DP, DT, которыми можно явно указывать 32-битовые и более числа).

К простейшим константным выражениям относятся:

- числа от $-2^{31}$ до $2^{32}-1$;
- символ (значением такого выражения является код символа);
- строка из двух символов (значением является слово-число, составленное из кодов этих символов);

- имя константы (значением такого выражения является значение константы).

К константным выражениям применимы следующие арифметические операторы (*k*, *k1* и *k2* означают любые константные выражения):

- одноместные плюс и минус *+k*, *–k*;
- операторы сложения и вычитания *k1+k2*, *k1–k2*;
- оператор умножения *k1*k2*, целочисленного деления *k1*/*k2*, получения остатка от деления *k1* MOD *k2*.

*Пример:*
```
K  EQU   30
X  DB    (3*K-1)/2 DUP(?)     ;массив из 44 байтов
```

Операндами арифметических операторов должны быть константные выражения. Язык ассемблер разрешает вычитать один адрес из другого, в результате Вы получите число, а не адрес.

*Пример:*
```
X         DW     1,2,3,4,5
Y         DB     ?
SIZE_X    EQU    Y-X          ;SIZE_X=10
```

Вычитание адресов используется обычно для определения расстояния (числа байтов) между этими адресами. Результатом действия арифметических операторов на константные выражения будет 32-битное число. Например: 2*80000000h=00000000h а не 100000000h.

### *Адресные выражения*

Значениями адресных выражений являются 32-битовые адреса.
К простейшим адресным выражениям относятся:

- метка (имя команды) и имя переменной, описанное в директиве DB, DW или DD (значениями таких выражений являются адреса меток и имен);
- счетчик размещения; он записывается как $ и обозначает адрес того предложения, в котором он встретился. В разных предложениях $ обозначает разные адреса. Например, если адрес переменной A равен 400100h, то имеем:

```
A  DD   $    ;эквивалентно     A       DD        400100h
B  DD   $    ;эквивалентно     B       DD        400104h
```

Чаще всего счетчик размещения используется для вычисления размера памяти занимаемой каким-то массивом.

*Пример:*
```
X DW 40 DUP(?)
SIZE_X EQU $-X      ;SIZE_X = 80
```

Здесь $ обозначает адрес первого байта за массивом X, из этого адреса и вычитается начальный адрес массива.

## Мнемоника команд MMX, SSE, SSE2

Когда разберёшься как по какому принципу строятся мнемонические команды – тогда всё станет легко. Вот что присутствует в мнемонике:

***SIMD для работы с вещественными числами:***

[h] – horizontal операции
op – код операции
[op] – код операции2 для сложных инструкций
[L|h] – обозначает какая часть приёмника/источника подвержена операции op: low|high
[[L|h]] – если присутствует, то обозначает куда будет помещён результат операции op в зависимости от предыдущей мнемоники. Возможно: L->h, h->L
[a|u] – aligned|unaligned. Говорит о требованиях к выравниванию операнда(ов) в памяти: a – выравнивание на границу в 16 байт требуется, u – не требуется
[nt] – non temporal. Говорит о некэшируемости операндов в памяти
[s|p] – scalar/packed операция op над данными (для h префикса только p)
s|d – single/double precision. Размерность данных в операндах. (float/double)
[2] – «to». Возникает при операциях op конвертации.
[s|p] – scalar/packed вид данных операнда. Возникает 2-й раз в операциях конвертации после «2»
[s|d] – single/double precision. Размерность данных в операндах. (float/double). Возникает 2-й раз в операциях конвертации после «2» и вида данных операции op
[[h|L]] – high|low. Присутствует, если используется мнемоника DUP (смотри ниже). Указывает расположение данных в операндах источниках для выполнения операции op
[dup] – duplicate. Может возникнуть при дублировании частей операндов источников в операнде приёмнике после выполнении операции op

*Примеры*: addps, haddps, addsubpd, movhlps, andps, movaps, movntpd, movshdup...

***SIMD для работы с целыми числами:***

[p] – packed. Присутствует всегда за исключением малого количества некоторых операций op
[h] – horizontal операции
op – код операции

[op] – код операции2 в сложных операциях

[s|us] – signed|unsigned saturation. Тип насыщение при операции op: знаковое|беззнаковое. Если отсутствует, то без насыщения (wrap around арифметика)

[L|h] – low|high. Обозначает, какая часть приёмника/источника подвержена операции op

[nt] – non temporal. Говорит о некэшируемости операндов в памяти

[b|w|d|q|dq] – byte|word|dword|qword|dqword. Размер упакованных операндов источников

[[2]] – «to». Может возникнуть при операции op над данными при разных типах операндов источника и приёмника: MMX и XMM регистров. «2» говорит о направлении действия операции op над данными. Возможно: Q2DQ, DQ2Q

[[b|w|d|q|dq]] – byte|word|dword|qword|dqword. Размер упакованных данных операции op. Осуществляется преобразование от меньшей разрядности операций (смотри предыдущую мнемонику) к большей (см. эту). Возможно от b->w, w->d, d->q... Если отсутствует, то размер результата упакованных данных остаётся неизменным (т.е. см. предыдущую мнемонику)

[a|u] – aligned|unaligned. Говорит о требованиях к выравниванию операнда(ов) в памяти: a – выравнивание на границу в 16 байт требуется, u – не требуется

*Примеры*: paddb, phsubw, psubusb, punpckhbw, pand, movntdqa, movdqu...

## Контрольные вопросы

1. Какие из следующих имен не могут быть идентификатором: а) C, б) $50, в) @$_Z, г) 34B7, д) AX?
2. Какова длина в байтах для элементов данных, определенных директивами: а) DW, б) DD, в) DT, г) DB, д) DQ?
3. Перечислите допустимые спецификаторы, которые могут встречаться при описании чисел в ассемблере.
4. Определите символьную строку по имени TITLE1, содержащую константу RGB Electronics.
5. Является ли запись A5h правильным шестнадцатеричным числом?
6. Как временно исключить какой-либо фрагмент программы из общего текста программы не удаляя этот фрагмент?
7. Запишите константное выражение, в котором вычисляется остаток от деления 10 на 3.
8. Почему при написании программ на языке ассемблер не стоит использовать числовые адреса памяти для доступа к переменным?

9. Определите следующие числовые значения в элементах данных от FLDA до FLDE: а) четырёхбайтного элемента, содержащего шестнадцатеричный эквивалент десятичного числа 115, б) однобайтового элемента, содержащего шестнадцатеричный эквивалент десятичного числа 25, в) двухбайтового элемента, содержащего неопределенное значение, г) однобайтового элемента, содержащего двоичный эквивалент десятичного числа 25, д) директивы DW, содержащей последовательные значения 16, 19, 20, 27, 30.
10. Приведите пример правильной вещественной константы, содержащей показатель степени.
11. Нужно ли заключать строковую константу в одинарные кавычки?
12. В чем разница между DB ‘26’ и DB 26.
13. Назовите максимальную длину идентификатора.
14. Определите ассемблерный шестнадцатеричный объектный код для: а) DB –26, б) DW 2645, в) DD 25733, г) DQ –2573300.
15. Представьте следующие символьные цепочки в шестнадцатеричном коде: а) SAM JONES; б) –75.61; в) Hello, how are you?
16. Верно ли, что название идентификаторов в языке ассемблер не зависят от регистра символов?
17. Приведите пример многострочного комментария.
18. Как сохраняется в памяти строка символов? Как подсчитать размер занимаемой памяти для отдельной строки?
19. Верно ли, что для записи директив можно использовать как прописные, так и строчные латинские буквы, а также их сочетание?
20. Что такое константа? Чем константа отличается от переменной?
21. Назовите основные части ассемблерной команды.

    Верно ли, что метка в коде программы должна заканчиваться двоеточием, а метка данных нет?

ГЛАВА 7

# КОМАНДЫ ПЕРЕДАЧИ ДАННЫХ

## 7.1. Команды пересылки

Среди команд микропроцессора семейства x86 достаточно много команд пересылки. Здесь мы рассмотрим следующие: MOV, LEA, XCHG и BSWAP. Кроме того, рассмотрим оператор PTR.

### 7.1.1. Команда MOV

(пересылка =”**MOV**E operand”)

Команда пересылки байта, слова или двойного слова. Пересылаемая величина берется из команды, регистра или ячейки памяти, а записывается в регистр или ячейку памяти. Таких команд много, но в языке ассемблера все они записываются одинаково: MOV <*DEST*>,< *SRC*>

*Алгоритм работы:* копирование второго операнда в первый операнд.

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| LDDQU | | | | | | | | | ✓ | ✓ |
| MOV | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MOVAPD | | | | | | | ✓ | ✓ | ✓ | ✓ |
| MOVAPS | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| MOVD | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MOVDQA | | | | | | | ✓ | ✓ | ✓ | ✓ |
| MOVDQU | | | | | | | ✓ | ✓ | ✓ | ✓ |
| MOVQ | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| MOVUPD | | | | | | | ✓ | ✓ | ✓ | ✓ |
| MOVUPS | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |

*Возможные варианты команды*:

```
        mov   reg/mem, reg
        mov   reg, mem
        mov   mem/reg(8/16/32/64), imm(8/16/32)
        mov   sreg, m/r16
        mov   m/r16, sreg
MMX     movd  mmx,m64/mmx
        movq  mmx,m64/mmx
SSE     movups xmm,m128/xmm
        movups m128,xmm
        movaps xmm,m128/xmm
        movaps m128,xmm
SSE2    movdqu xmm,m128/xmm
```

```
        movdqu m128,xmm
        movupd xmm,m128/xmm
        movupd m128,xmm
        movdqa xmm,m128/xmm
        movdqa m128,xmm
3DNow!  movq mmx,m64/mmx
        lddqu xmm,m128
```

*Псевдокод команды:* *DST←SRC*

*Применение:* команда MOV применяется для различного рода пересылок данных, при этом, несмотря на всю простоту этого действия, необходимо помнить о некоторых ограничениях и особенностях выполнения данной операции:

- направление пересылки в команде MOV всегда справа налево, то есть из операнда *SRC* в операнд *DEST*;
- значение операнда *SRC* не изменяется;
- оба операнда не могут быть из памяти (при необходимости можно использовать цепочечную команду MOVS или сочетание PUSH/POP);
- лишь один из операндов может быть сегментным регистром;
- лишь один из операндов может быть управляющим регистром;
- лишь один из операндов может быть тестовым регистром;
- лишь один из операндов может быть регистром отладки;
- лишь один из операндов может быть непосредственным значением;
- желательно использовать в качестве одного из операндов регистр AL/AX/EAX/RAX, так как в этом случае транслятор генерирует более короткую форму команды MOV.

*Примеры:*

```
MOV AL,5;непосредственная запись байта в регистр
MOV BL,AL;пересылка байта из регистра в регистр
MOV Omega,CX ;пересылка слова из регистра в ячейку памяти
MOV EAX,0FFFFFFFFh;непосредственная запись двойного слова в регистр
MOV BX,DS ;пересылка байта из сегментного регистра в регистр
                ;общего назначения
```

Команда MOV применяется для обмена данными между системными регистрами. Это одна из немногих возможностей доступа к содержимому этих регистров. Данную команду можно использовать только на нулевом уровне привилегий либо в реальном режиме работы микропроцессора.

```
MOV EAX,CR0;переключение микропроцессора в защищенный режим
OR EAX,1 ;помещаем в нулевой бит EAX 1
MOV CR0,EAX
```

Если необходимо переслать в регистр адрес какой-то переменной, то Вам придется использовать в команде MOV оператор OFFSET (оператор получения смещения выражения). OFFSET позволяет получить значение смещения выражения в байтах относительно начала того сегмента, в котором выражение определено. А если эта переменная еще и находится в другом сегменте, то в паре с оператором OFFSET Вам придется использовать оператор SEG (оператор получения сегментной составляющей адреса выражения). Оператор SEG возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.

Например, если в сегменте данных содержится POLE, то следующие команды пересылают в пару ES:EDX полный адрес этой переменной:

```
MOV AX,SEG POLE
MOV ES,AX
MOV EDX,OFFSET POLE
```

Здесь и далее будут показаны эквиваленты команд с целью показать вам либо более короткий код (используется при оптимизации программ по размеру), либо более длинный код (используется при написании самомодифицируемого кода), либо фрагменты кода, приводящие к эквивалентному результату (можно «размазать» данный код по программе, чтобы спрятать алгоритм от «кодокопателей»). В общем, автор показывает возможность, а уж как вы это будете использовать, это уже ваше дело...

| Команда | Эквивалент |
|---|---|
| MOV EAX,imm | LEA EAX,[imm] |
| MOV EAX,0 | XOR EAX,EAX/SUB EAX,EAX/AND EAX,0/IMUL EAX,0 |
| MOV [imm],0 | Если вы знаете, что регистр EAX=0, то MOV [imm],EAX |
| MOV EAX,imm | Если вы знаете, что регистр EBX=0, а число imm находится в диапазоне от -128 до +127, то LEA EAX,[EBX+imm] |
| MOV EAX,0FFFFFFFFh | MOV не делает знаковое расширение непосредственного оператора, а логические операторы делают, поэтому код MOV EAX,-1 длиннее чем OR EAX,-1 |
| MOV EDX,0 | CWDE если EAX<80000000h |
| MOV ECX,0 | a1: loop a1 |

## Программирование на уровне битов

Для начала учим наизусть таблицу 7.1.1 с номерами регистров:

*Таблица 7.1.1*

**Бит W и кодировка регистров**

| dec | bin | W=0 | W=1 | | Segment | Control | Debug | Test |
|---|---|---|---|---|---|---|---|---|
| | | rb | rw | rd | | | | |
| 0 | 000 | AL | AX | EAX | ES | CR0 | DR0 | |
| 1 | 001 | CL | CX | ECX | CS | | DR1 | |
| 2 | 010 | DL | DX | EDX | SS | CR2 | DR2 | |
| 3 | 011 | BL | BX | EBX | DS | CR3 | DR3 | TR3 |
| 4 | 100 | AH | SP | ESP | FS | CR4 | | TR4 |
| 5 | 101 | CH | BP | EBP | GS | | | TR5 |
| 6 | 110 | DH | SI | ESI | | | DR6 | TR6 |
| 7 | 111 | BH | DI | EDI | | | DR7 | TR7 |

При попытке использовать несуществующие регистры будет сгенерировано исключение invalid opcode.

| Инструкция | Bin-код | Hex-код | Мнемоника |
|---|---|---|---|
| Поместить число imm в регистр | 1011 w reg: imm | B0+rb | mov r8,imm8 |
| | | B8+rw | mov r16,imm16 |
| | | B8+rd | mov r32,imm32 |
| Поместить число imm в регистр (альтернативная кодировка) | 1100 011w:11 000 reg: imm | C6 | mov r8,imm8 |
| | | C7 | mov r16,imm16 |
| | | C7 | mov r32,imm32 |
| Поместить число imm в ячейку памяти mem | 1100 011w:mod 000 r/m: imm | C6 | mov m8,imm8 |
| | | C7 | mov m16,imm16 |
| | | C7 | mov m32,imm32 |

Мы хотим поместить в EAX число 1. Для этого к опкоду добавляем номер регистра (EAX=000).

1011:w:reg:imm→1011:1:000:imm=0B8:imm

Число, которое надо поместить в регистр, пишем сразу после опкода. То есть кодировка выглядит следующим образом:

```
db      0B8h
dd      1 ; mov eax,1
```

| | опкод | w | Reg | imm32 |
|---|---|---|---|---|
| bin | 1011 | 1 | 000=EAX | 00000001.00000000.00000000.00000000 |
| hex | 0B8h | | | 1,0,0,0 |

Обратите внимание: единица вставлена с помощью директивы dd, которую можно заменить на последовательность байтов 1,0,0,0. В этой по-

следовательности байт с единицей идёт первым, так как в памяти байты слов и двойных слов располагаются от младшего к старшему, а единица находится в младшем байте. Вышеприведённую команду можно переписать так:

```
db  0B8h,1,0,0,0; mov eax,1
```

А можно и так, хотя это на 1 байт длиннее:

```
db  0C7h,0C0h,1,0,0,0 ; mov eax,1
```

| | **опкод** | **w** | **mod** | **opcode** | **Reg/Mem** | **imm32** |
|---|---|---|---|---|---|---|
| bin | 1100011 | 1 | 11 | 000 | 000=EAX | 00000001.00000000.00000000.00000000 |
| hex | 0C7h | | 0C0h | | | 1,0,0,0 |

Инструкция mov mem/reg,imm имеет следующий формат:

1100 011:w:Mod:000:r/m [SIB и/или смещение] imm.

В зависимости от значений в ModR/M в этот блок помимо байта ModR/M могут быть включены последовательно байт SIB и/или смещение. Обычно этот формат используется только для `mov mem,imm` но ничто не мешает использовать его и для кодирования `mov reg,imm`, поместив в поле Mod=11. При этом этот вариант будет на байт длиннее, чем команда в формате 1011:w:reg imm.

Лишний байт можно использовать в целях выравнивания для того, чтобы обойтись без дополнительных команд и тактов процессора. Поле Reg/Opcode=000. В противном случае возникнет исключение invalid opcode.

Вместо единицы можно подставить любое 32-битное число *imm*. Число *imm* будет занимать столько байт сколько и приемник *reg*.

Определение размера *reg* происходит так же, как и в других случаях, при помощи бита w. Если w = 0, то размер reg 1 байт и трёхбитовое поле *reg* определяет один из восьми 8-битных регистров. Если w = 1, то размер операнда слово или двойное слово (Таблица 7.1.1).

Наличие префикса 66h переключает команду между полным и альтернативным полным размером (при размере по умолчанию 16 бит – делает операнд 32-битным и наоборот). Например, код 8BC2h соответствует сразу двум командам `mov ax,dx` и `mov eax,edx`. Какой вариант выберет процессор будет зависеть от того, в каком режиме работает программа. Если в 32-битном режиме, то, встретив опкод 8BC2h, процессор выполнит инструкцию `mov eax,edx`. Если в 16-битном, то `mov ax,dx`. Чтобы выполнить команду `mov ax,dx` в 32-битном режиме, непосредственно перед инструкцией помещаем префикс 66h. Таким образом, код команды `mov ax,dx` в 32-битном режиме будет выглядеть так:

```
.386
. . . .
```

```
db 66h,8Bh,0C2h ;mov ax,dx
```

Тот же префикс в 16-битном режиме выполняет обратную работу – заставляет процессор выполнить данную инструкцию как 32-битную.

```
.286
. . . .
db 66h,8Bh,0C2h ;mov eax,edx
```

Если нужно поместить какое-либо число не в EAX, а в EDI, тогда к базовому опкоду (0B8h) следует прибавить номер регистра (EDI = =111b=7).

1011:w:reg→1011:1:111b=0B8h+7=0BFh

```
db 0B9h,50h,0,0,0;mov ecx,50h
```

| | **опкод** | **w** | **Reg** | **imm32** |
|---|---|---|---|---|
| bin | 1011 | 1 | 001=ECX | 00000101.00000000.00000000.0000000 |
| hex | 0B9h | | | 50h,0,0,0 |

```
db 0BAh,20h,0,0,0;mov edx,20h
```

| | **опкод** | **w** | **Reg** | **imm32** |
|---|---|---|---|---|
| bin | 1011 | 1 | 010=EDX | 00000010.00000000.00000000.00000000 |
| hex | 0BAh | | | 20h,0,0,0 |

От копирования числа в регистры общего назначения переходим к копированию содержимого регистра в регистр.

В языке ассемблера для копирования непосредственного значения *imm* в регистр, как и содержимого регистра в другой регистр или в память, применяется одна мнемоника – MOV. В тоже время эти варианты имеют разные опкоды. Причем зачастую к этим опкодам применяются разные правила.

Начнём с кодировки инструкции `mov eax,edx`. Базовый опкод инструкции `mov reg1,reg2` – 8Bh, но здесь уже два операнда, а не один.

Для составления требуемого кода используется дополнительный байт, который называется ModR/M, он располагается сразу после опкода. Сам ModR/M делится на три поля следующим образом:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Mod | | Reg/Opcode | | | R/M | | |

Биты 3-5 (поле Reg/Opcode) могут представлять либо уточняющий опкод (в случае, если один из операндов представлен непосредственным значением), либо регистр. Поля Reg/Opcode (пока назовем это поле просто Reg) и R/M служат для указания операндов.

Составим код команды – в Reg помещаем номер регистра *reg1*, в R/M – номер *reg2*. Поле Mod равно 3. Вот и сам код:

```
db 8Bh,0C2h;mov eax,edx
```

| | **опкод** | **d** | **w** | **Mod** | **Reg** | **R/M** |
|---|---|---|---|---|---|---|
| bin | 100010 | 1 | 1 | 11 | 000=EAX | 010=EDX |
| hex | 8Bh | | | 0C2h | | |

Формулы этих инструкций, когда операндами являются два регистра такова:

- mov reg1,reg2 → 1000 100w:11 reg1 reg2
- mov reg1,reg2 → 1000 101w:11 reg2 reg1

Используя эти формулы легко вычислить, например, что у команды `mov eax,edx` есть две кодировки:

1000100:w:11 reg1 reg2→1000:100:1:11:010:000b=89D0h и
1000101:w:11 reg2 reg1→1000:101:1:11:000:010b=8BC2h

```
db 8Bh,0C2; mov eax,edx
```

| | **опкод** | **d** | **w** | **Mod** | **Reg** | **R/M** |
|---|---|---|---|---|---|---|
| bin | 100010 | 0 | 1 | 11 | 000=EAX | 010=EDX |
| hex | 89h | | | 0C2h | | |

```
db 89h,0D0h;mov eax,edx
```

| | **опкод** | **d** | **w** | **Mod** | **Reg** | **R/M** |
|---|---|---|---|---|---|---|
| bin | 100010 | 1 | 1 | 11 | 010=EDX | 000=EAX |
| hex | 8Bh | | | 0D2h | | |

Бит d называется *битом направления*. Бит d показывает, чем является регистр в поле Reg: операндом-источником (d=0) или операндом-приемником (d=1)

Кодировку инструкции можно заменить на более универсальную:

mov reg1,reg2 → 100010:d:w:11:reg1:reg2

## Режимы адресации

Способ определения местонахождения операнда называется *режимом адресации*. Операнд машинной команды может находится в регистре процессора, указываться непосредственно в инструкции или находится в памяти данных. Процессоры семейства x86 поддерживают следующие режимы адресации:

- *Регистровая адресация:*

```
mov eax,edx
```

В регистр EAX скопировать значение, которое находится в регистре EDX.

- *Непосредственная* (прямая) *адресация:*

```
mov eax,[401000h]
```

В регистр EAX скопировать значение, которое находится в ячейке памяти с адресом 401000h. Адрес ячейки явно указан в самой инструкции.

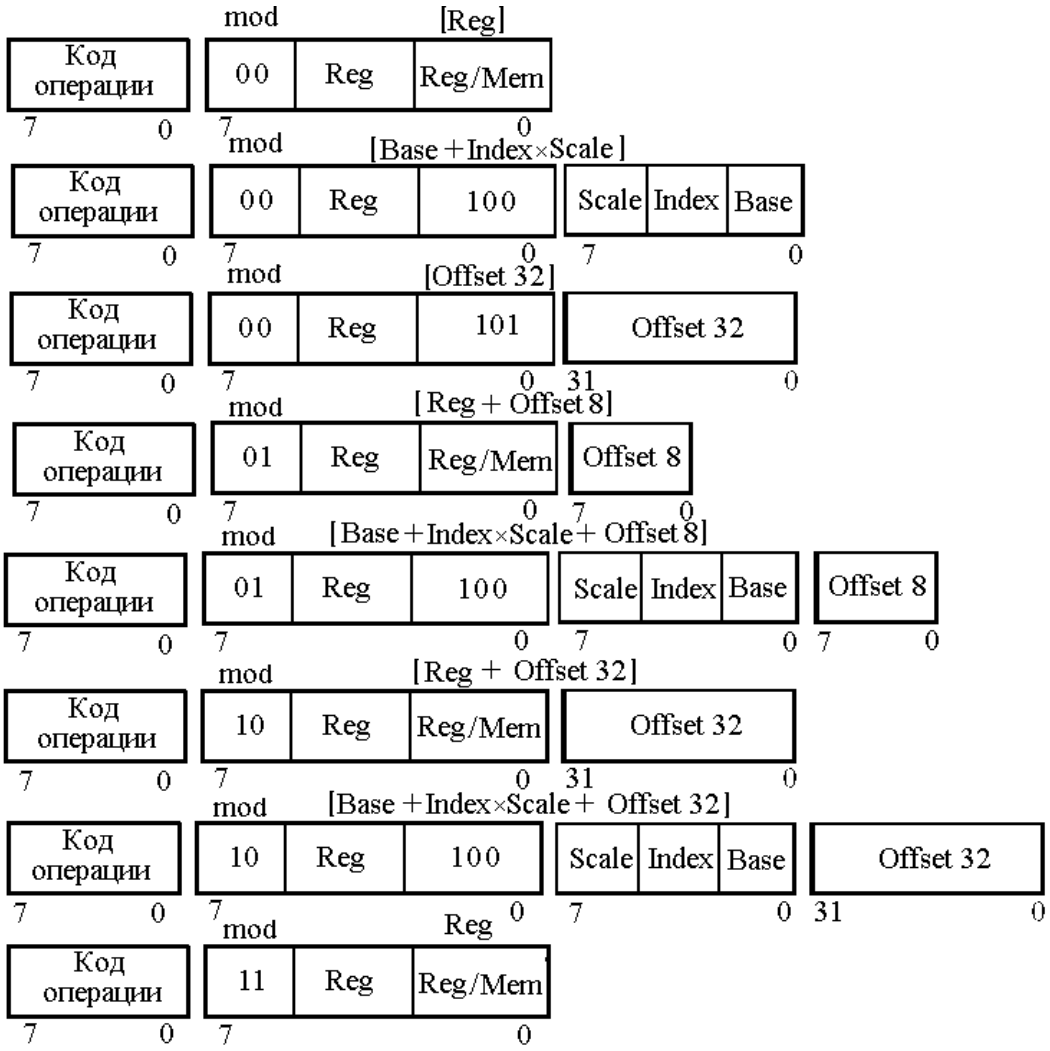



***Рис. 7.7.1. Формат команды в зависимости от поля MOD***

Адрес данных может быть задан косвенно — в команде указывается местоположение элемента памяти, где находится адрес операнда

- *Косвенная базовая адресация:*

```
mov eax,[edx]
```

В регистр EAX скопировать значение ячейки памяти, адрес которой находится в регистре *Base* (=EDX).

- *Косвенная базовая адресация со смещением:*

```
mov eax,[edx+1000h]
```

В регистр EAX скопировать значение ячейки памяти, адрес которой равен сумме 32-битного смещения (=1000h) и содержимого регистра *Base* (=EDX).

- *Базово-индексная адресация:*

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

В регистр EAX копируется значение ячейки памяти, адрес которой равен сумме содержимого регистра *Base* (=EDX) и содержимого регистра *Index* (=ECX).

- *Относительная базово-индексная адресация:*

```
mov eax,[edx+ecx+1000h]
```

В регистр EAX копируется содержимое ячейки памяти, адрес которой вычисляется следующим образом: содержимое регистра *Index* (=ECX) прибавляется к содержимому регистра *Base* (=EDX), а к полученному результату добавляется *смещение* (=1000h).

- *Относительная базово-индексная адресация с масштабированием:*

```
mov eax,[edx+ecx*4+1000h]
```

В регистр EAX копируется содержимое ячейки памяти, адрес которой вычисляется следующим образом: содержимое регистра *Index* (=ECX) умножается на *Scale* (может быть 1, 2, 4 и 8, в данном случае *Scale*=4), прибавляется к содержимому регистра *Base* (=EDX), а к полученному результату добавляется *смещение* (=1000h).

Косвенная адресация может быть только в одном операнде, но не в двух. Поэтому инструкции, подобные следующей, недопустимы, а при попытке их трансляции будет выдана ошибка:

```
mov [eax],[edx]
```

Поле Mod задаёт, что именно кодируется в Reg/Mem (рисунок 0). В сочетании с Mod, Reg/Mem может кодировать не только регистры, но и различные режимы косвенной адресации (таблица 7.1.2). Назначение поля Reg при этом не изменяется – в нём всегда кодируется номер регистра (либо расширение опкода).

*Таблица 7.1.2*

**Зависимость режима адресации от поля Mod**

| Mod | Адрес |
|---|---|
| 0 | [Reg] |
| 1 | [Reg+8-битное смещение] |
| 2 | [Reg+32-битное смещение] |
| 3 | Reg |

Двухбитовое поле Mod вместе с трехбитовым полем Reg/Mem образует $2^5$=32 возможных значения, 8 из которых обозначения регистров и 24 соответствуют режимам адресации.

Если Mod=3 в R/M находится номер регистра.

| R/M | Адрес | | | | |
|---|---|---|---|---|---|
| 000 | AL | AX | EAX | RAX | |
| 001 | CL | CX | ECX | RCX | |
| 010 | DL | DX | EDX | RDX | |
| 011 | BL | BX | EBX | RBX | |
| 100 | AH | SP | ESP | RSP | SPL |
| 101 | CH | BP | EBP | RBP | BPL |
| 110 | DH | SI | ESI | RSI | SIL |
| 111 | BH | DI | EDI | RDI | DIL |

Вот так интерпретируется значение поля ModR/M в режиме Mod=0.

| R/M | Адрес | SIB |
|---|---|---|
| 000 | DS: [EAX] | не представлен |
| 001 | DS:[ECX] | не представлен |
| 010 | DS: [EDX] | не представлен |
| 011 | DS:[EBX] | не представлен |
| 100 | за байтом ModR/M следует байт SIB | |
| 101 | DS:[32-битное смещение] | не представлен |
| 110 | DS:[ESI] | не представлен |
| 111 | DS:[EDI] | не представлен |

Составим кодировку для инструкции:

```
mov eax,[edx]
```

8Bh это опкод операции *mov reg32, mem32*:

```
db 8Bh,2 ;mov eax,[edx]
```

| | опкод | d | w | Mod | Reg | R/M |
|---|---|---|---|---|---|---|
| bin | 1000 10 | 1 | 1 | 00 | 000=EAX | 010=[EDX] |
| hex | 8B | | | 02 | | |

А теперь манипулируем битами d и w

```
db 8Ah,2 ;mov al,[edx]
```

| | опкод | d | w | Mod | Reg | R/M |
|---|---|---|---|---|---|---|
| bin | 1000 10 | 1 | 0 | 00 | 000=AL | 010=[EDX] |
| hex | 8A | | | 02 | | |

```
db 89h,2 ;mov [edx],eax
```

| | опкод | d | w | Mod | Reg | R/M |
|---|---|---|---|---|---|---|
| bin | 1000 10 | 0 | 1 | 00 | 000=EAX | 010=[EDX] |
| hex | 89h | | | 02 | | |

```
db 88h,2 ;mov [edx],al
```

| | опкод | d | w | Mod | Reg | R/M |
|---|---|---|---|---|---|---|
| bin | 1000 10 | 0 | 0 | 00 | 000=AL | 010=[EDX] |
| hex | 88h | | | 02 | | |

Пусть у нас есть переменная *var*, содержимое которой нужно скопировать в ESI. Чтобы закодировать команду `mov esi,var`, помещаем в поле R/M значение 101b, а после байта ModR/M – адрес переменной:

```
db 8Bh,35h;mov esi,offset var
dd offset var
```

| | опкод | d | w | Mod | Reg | R/M | imm32 |
|---|---|---|---|---|---|---|---|
| bin | 1000 10 | 1 | 1 | 00 | 110=ESI | 101 | var |
| hex | 8Bh | | | 35h | | | var |

Обратите внимание, в списке отсутствуют регистры EBP и ESP, хотя, инструкции типа `mov eax,[ebp]` и `mov eax,[esp]` поддерживаются. Для инструкций такого типа используют Mod=1. В этом режиме R/M интерпретируется следующим образом:

| R/M | Адрес | SIB |
|---|---|---|
| 000 | DS:[EAX+ смещение8] | не представлен |
| 001 | DS: [ECX+ смещение8] | не представлен |
| 010 | DS: [EDX+ смещение8] | не представлен |

| R/M | Адрес | SIB |
|---|---|---|
| 011 | DS: [EBX+ смещение8] | не представлен |
| 100 | за ModR/M следует SIB, а далее 8-битное смещение | |
| 101 | SS: [EBP+ смещение8] | не представлен |
| 110 | DS:[ESI+ смещение8] | не представлен |
| 111 | DS: [EDI+ смещение8] | не представлен |

Из этого списка можно получить ответ – как же закодировать инструкцию MOV EDI,[EBP]. Фактически, у нас получается MOV EDI, [EBP+0], а кодировка будет следующей:

```
db 8Bh,01111101b,0; MOV EDI,[EBP]
```

| | опкод | d | w | Mod | reg | R/M | смещение8 |
|---|---|---|---|---|---|---|---|
| bin | 1000 10 | 1 | 1 | 01 | 111=EDI | 101=[EBP+смещение8] | 00000000 |
| hex | 8Bh | | | 35h | | | 0 |

Интерпретация значений поля ModR/M в режиме Mod=2

| R/M | Адрес | SIB |
|---|---|---|
| 000 | DS:[EAX + смещение32] | не представлен |
| 001 | DS: [ECX+ смещение32] | не представлен |
| 010 | DS: [EDX+ смещение32] | не представлен |
| 011 | DS: [EBX+ смещение32] | не представлен |
| 100 | за ModR/M следует SIB, а далее 32-битное смещение | |
| 101 | SS:[EBP+ смещение32] | не представлен |
| 110 | DS: [ESI+ смещение32] | не представлен |
| 111 | DS: [EDI+ смещение32] | не представлен |

Кодируем инструкцию `mov ecx,[ebx+100000h]`:

```
db 8Bh,8Bh
dd 100000h ;32-битное смещение
```

| | опкод | d | w | Mod | Reg | R/M | 32-битное смещение |
|---|---|---|---|---|---|---|---|
| bin | 100010 | 1 | 1 | 10 | 001=ECX | 011=[EBX+ смещ32] | 00000000.00000000.00010000.00000000 |
| hex | 8Bh | | | 8Bh | | | 0, 0, 10h, 0 |

Итак, теперь вы умеете кодировать инструкции, в которых есть косвенная адресация вида [регистр+смещение] или просто адрес переменной. Чтобы окончательно понять инструкции ассемблера и овладеть искусством кодирования во всех режимах адресации, потребуется разобраться со следующим полем, которое применяется

для этих целей – *SIB* (**S**cale**I**ndex **B**ase). С помощью байта SIB можно задавать выражения вида:

[*Base* + *Index* × *Scale* +смещение].

Регистр *Base* – это один из 8 регистров общего назначения. Регистр *Index* – тоже один из регистров общего назначения, за исключением ESP. Множителем *Scale* может быть число 1, 2, 4 или 8. *Смещение* может быть 8-, 16- или 32-битным (последние два – в зависимости от того, в каком режиме работает программа, в 16- или 32-битном). С помощью префикса 67h можно в 16-разрядном режиме использовать 32-разрядные режимы адресации, и наоборот.

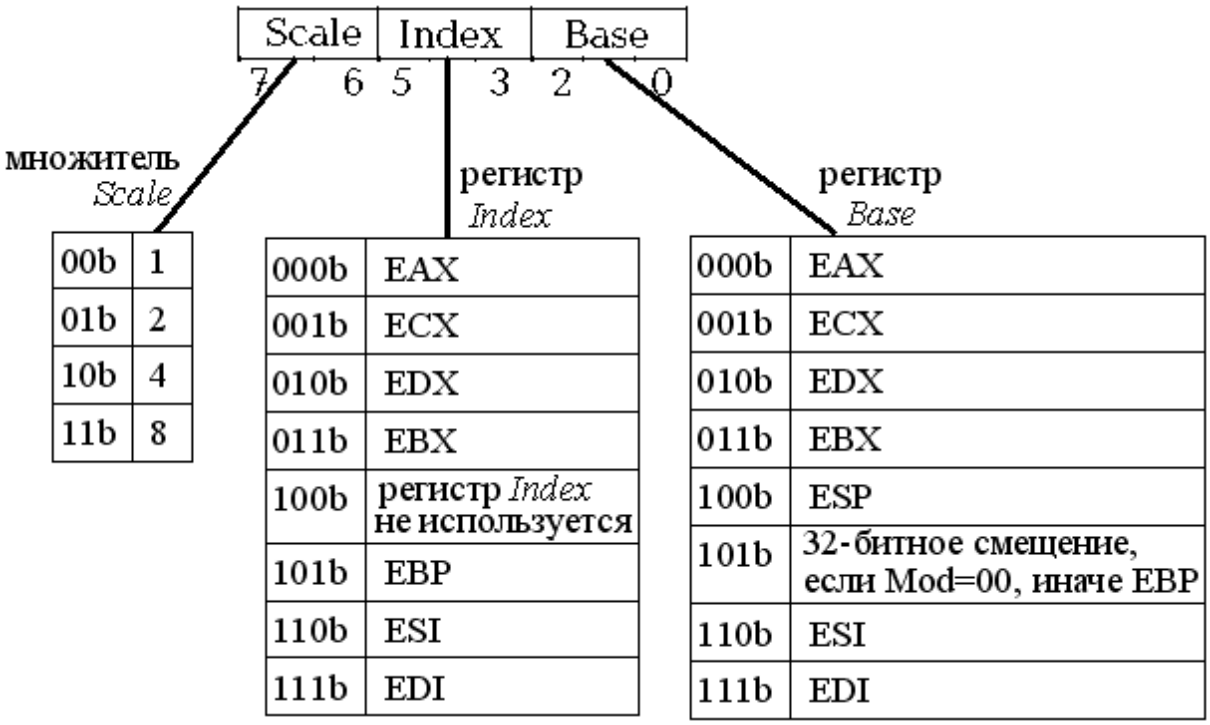



***Рис. 7.1.2. Структура байта SIB***

Закодируем инструкцию `mov ecx,[edi+esi*4]`. Разберем ModR/M: Mod =0, Reg=001b (ECX), R/M=100b (потому что далее следует SIB). Теперь разберем SIB: *Scale*=10 (×4), регистр *Index* (который должен быть умножен на 4) =110b (ESI), регистр *Base*=111b (EDI).

В итоге получается код:

```
db 8Bh,0Ch,0B7h;mov ecx,[edi+esi*4]
```

| | **опкод** | **d** | **w** | **Mod** | **Reg** | **R/M** | ***Scale*** | ***Index*** | ***Base*** |
|---|---|---|---|---|---|---|---|---|---|
| bin | 100010 | 1 | 1 | 00 | 001=ECX | 100 | 10=4 | 110=ESI | 111=EDI |
| hex | 8Bh | | | 0Ch | | | 0B7h | | |

Кодируем инструкцию `mov eax,[esp]`. Mod=0, Reg=000b (EAX), R/M=100b (понадобится поле SIB). Байт SIB: множитель *Scale*=0 (×1), *Index*=100b (не используется), *Base*=100b (ESP):

```
db 8Bh,4,24h;mov eax,[esp]
```

| | **опкод** | **d** | **w** | **Mod** | **Reg** | **R/M** | ***Scale*** | ***Index*** | ***Base*** |
|---|---|---|---|---|---|---|---|---|---|
| bin | 100010 | 1 | 1 | 00 | 000=EAX | 100 | 00=1 | 100 | 100=ESP |
| hex | 8Bh | | | 4 | | 24h | | | |

Изменяя значение полей Mode и SIB, получаем всевозможные варианты одной и той же инструкции `mov eax,[esi]`:

| **Код** | **Команда** |
|---|---|
| 8B 06 | mov eax,[esi] |
| 8B 04 35 00 00 00 00 | mov eax,[*Index*=esi×1+смещ32=0] |
| 8B 04 26 | mov eax,[*Base*=esi] |
| 8B 46 00 | mov eax,[esi+смещ8=0] |
| 8B 44 26 00 | mov eax,[*Base*=esi+смещ8=0] |
| 8B 86 00 00 00 00 | mov eax,[esi+смещ32=0] |
| 8B 84 26 00 00 00 00 | mov eax,[*Base*=esi+смещ32=0] |

А теперь итоговая таблица

| **R/M** | **Mod** | | | | **SIB** | |
|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **Index** | **Base** |
| 000 | [EAX] | [EAX+смещ8] | [EAX+смещ32] | EAX | [EAX*N] | EAX |
| 001 | [ECX] | [ECX+смещ8] | [ECX+смещ32] | ECX | [ECX*N] | ECX |
| 010 | [EDX] | [EDX+смещ8] | [EDX+смещ32] | EDX | [EDX*N] | EDX |
| 011 | [EBX] | [EBX+смещ8] | [EBX+смещ32] | EBX | [EBX*N] | EBX |
| 100 | за байтом ModR/M следует байт SIB | за ModR/M следует SIB, а далее 8-битное смещение | за ModR/M следует SIB, а далее 32-битное смещение | ESP | нет | ESP |
| 101 | за ModR/M следует 32-битное смещение (адрес памяти) | [EBP+смещ8] | [EBP+смещ32] | EBP | [EBP*N] | зависит от значения Mod |
| 110 | [ESI] | [ESI+смещ8] | [ESI+смещ32] | ESI | [ESI*N] | ESI |
| 111 | [EDI] | [EDI+смещ8] | [EDI+смещ32] | EDI | [EDI*N] | EDI |

Если в качестве базового регистра будет выбран EBP, то полученный режим адресации будет зависеть от значения Mod

| Mod | Действие |
|---|---|
| 0 | [*Index*+Смещение32] – регистр EBP в адресации не участвует |
| 1 | [EBP+*Index*+8-битное смещение] |
| 2 | [EBP+*Index*+32-битное смещение] |
| 3 | EBP |

Обратите внимание на следующее:

*во-первых*, регистр *Index* не может быть ESP, ведь значение 100b, которое должно было символизировать ESP, указывает, что *Index* не используется. Соответственно, закодировать инструкцию `mov eax,[edx+esp*4]` нельзя, а инструкции `mov eax,[esp+edx]` или `mov eax,[esp+4*edx]` можно;

*во-вторых*, если Mod=00b, то Reg1= 101b означает, что далее следует 8-битное смещение.

Поэтому, чтобы закодировать `mov eax,[ebp+edx]`, нужен описанный выше трюк: в Mod помещается 01b, а после SIB идет нулевое 8-битное смещение.

```
db 8Bh,44h,15h,0;mov eax,[ebp+edx+0]
```

| | опкод | d | w | Mod | Reg | R/M | *Scale* | *Index* | *Base* | offset8 |
|---|---|---|---|---|---|---|---|---|---|---|
| bin | 100010 | 1 | 1 | 01 | 000=EAX | 100 | 00=1 | 010=EDX | 101=EBP | 00000000 |
| hex | 8Bh | | | 44h | | | 15h | | | 0 |

### Уменьшение размера кодировки MOV

Там, где это возможно, меняйте адресацию с прямой на косвенную, но только в случае, когда смещение лежит в диапазоне от 80h(−128) до 7Fh(+127).

*Пример:* (Пусть EDI=5)

```
8B0F05          MOV EBX,[EDI+5]
```

код короче на 2 байта, чем

```
BB0A000000      MOV EBX,0Ah
```

Косвенная адресация с заранее известным значением в одном из регистров дает возможность использовать знаковое расширение непосредственного значения в команде MOV.

## 7.2. Команда LEA

(Загрузка эффективного адреса = “**L**oad **E**ffective **A**ddress”)

*Синтаксис команды*: LEA <*DEST*>,<*SRC*>

LEA <*DEST*>,[<*Base*>+<*Index*>*<*Scale*>+<*смещение*>]
где <*Scale*>*1,2,4 или 8*

*Возможные варианты команды:*

lea r(16/32/64), mem

*Семантика команды*: получить в регистр *DEST* эффективный адрес (смещение) операнда *SRC*.

*Алгоритм работы*: алгоритм работы команды зависит от действующего режима адресации (use 16/use 32 или use 64):

- если use 16, то в регистр *DEST* загружается 16-битное значение смещения операнда *SRC* ;
- если use 32, то в регистр *DEST* загружается 32-битное значение смещения операнда *SRC;*
- если use 64, то в регистр *DEST* загружается 64-битное значение смещения операнда *SRC*.

*Псевдокод команды* LEA:

```
IF OperandSize = 16 and AddressSize = 16
        THEN   DEST ← EffectiveAddress(SRC);  16-битная адресация
        ELSE IF OperandSize = 16 и AddressSize = 32
            THEN temp← EffectiveAddress(SRC);  32-битная адресация
                    DEST ← temp[0:15];  16-битная адресация
            ENDIF;
        ELSE IF OperandSize = 32 и AddressSize = 16
            THEN temp← EffectiveAddress(SRC);  16-битная адресация
                    DEST ← ZeroExtend(temp);  32-битная адресация
            ENDIF;
        ELSE IF OperandSize = 32 и AddressSize = 32
            THEN DEST ← EffectiveAddress(SRC);  32-битная адресация
            ENDIF;
        ELSE IF OperandSize = 16 и AddressSize = 64
            THEN temp← EffectiveAddress(SRC);  64-битная адресация
                    DEST ← temp[0:15];  16-битная адресация
            ENDIF;
        ELSE IF OperandSize = 32 и AddressSize = 64
            THEN temp← EffectiveAddress(SRC);  64-битная адресация
                    DEST ← temp[0:31];  16-битная адресация
            ENDIF;
        ELSE IF OperandSize = 64 и AddressSize = 64
            THEN DEST← EffectiveAddress(SRC);  64- битная адресация
            ENDIF;
```

ENDIF;

*Применение*: код эквивалентной команды mov ebx, offset adr короче кода lea ebx, adr

| Символическое представление инструкции | Машинный код инструкции | Ассемблерные команды |
|---|---|---|
| mov ebx, offset adr | BBF9004000 | mov ebx,0004000F9h |
| lea ebx, adr | 8D1DF9004000 | lea ebx,[004000F9h] |

Использование команды lea

| | |
|---|---|
| Для сложения | lea eax,[eax+ebx]=add eax,ebx |
| Для инкремента | lea eax,[eax+1] |
| Для декремента | lea eax,[eax-1] |
| Для умножения на 2 | lea eax,[eax+eax] или lea eax,[eax*2] |
| Для умножения на 3 | lea eax,[eax+eax*2] |
| Для умножения на 4 | lea eax,[eax*4] |
| Для умножения на 5 | lea eax,[eax+eax*4] |
| Для умножения на 6 | lea ecx,[eax*2] / lea eax,[ecx+ecx*2] |
| Для умножения на 7 | lea ecx,[eax+eax*2] / lea eax,[ecx+eax*4] |
| Для умножения на 8 | lea eax,[eax*8] |
| Для умножения на 9 | lea eax,[eax+eax*8] |

Команда `lea eax,[ebx]` эквивалент `mov eax,ebx`

## 7.3. Команда XCHG

(Обмен =”E**XCH**AN**G**E operands”)

В программах на языке ассемблера приходится довольно часто переставлять местами какие-то две величины, и хотя такую перестановку можно организовать с помощью двух команд MOV с сохранением значения в промежуточной ячейке памяти, в процессор введена специальная команда для этого.

*Синтаксис команды*: XCHG <*SRC*>,<*DEST*>

*Семантика команды:* обмен значений между двумя регистрами или между регистром и ячейкой памяти.

*Алгоритм работы:* обмен содержимым операнда *SRC* и операнда *DEST*.

*Псевдокод команды*:
*TEMP←DEST*
*DEST← SRC*
*SRC ←TEMP*

*Возможные варианты команды:*

xchg reg, mem/reg

*Применение: к*оманду XCHG можно использовать для выполнения операции обмена двух операндов с целью изменения порядка следования

байт, слов, двойных слов или их временного сохранения в регистре или памяти. Альтернативой является использование для этой цели стека или промежуточной ячейки памяти.

```
;поменять порядок следования байт в ячейке памяти
H1 DW 0F85Ch    ;напрямую командой XCHG нельзя, но
MOV AX,H1       ;можно для этой цели использовать
XCHG AH,AL      ;промежуточный регистр AX
MOV H1,AX       ;[H1]=5CF8h
```

Команду XCHG можно заменить на XOR или на PUSH/POP, или на MOV с использованием промежуточного регистра или промежуточной ячейки памяти.

Эквиваленты команды XCHG EAX,EBX

| | | |
|---|---|---|
| MOV [TEMP],EAX | XOR EAX,EBX | PUSH EAX |
| MOV EAX,EBX | XOR EBX,EAX | PUSH EBX |
| MOV EBX,[TEMP] | XOR EAX,EBX | POP EAX |
| | | POP EBX |

## 7.4. Команда обмена байтов BSWAP

(обмен байтами "**B**YTE **SWAP**ING")

*Синтаксис команды:* BSWAP *<DEST>*

*Возможные варианты команды:*

bswap reg(16/32/64)

*Семантика команды:* команда BSWAP изменяет порядок следования байтов в 32-разрядных регистрах, конвертируя значения из вида машинного представления (младшая часть, старшая часть) в обычное представление (старшая часть, младшая часть) и наоборот. Можно использовать для примитивного шифрования.

*Псевдокод команды* BSWAP:

```
    TEMP←DEST
IF 64-bit mode AND OperandSize = 64
    THEN
        DEST[7:0]←TEMP[63:56]
        DEST[15:8]←TEMP[55:48]
        DEST[23:16]←TEMP[47:40]
        DEST[31:24]←TEMP[39:32]
        DEST[39:32]←TEMP[31:24]
        DEST[47:40]←TEMP[23:16]
        DEST[55:48]←TEMP[15:8]
        DEST[63:56]←TEMP[7:0]
```

```
            ELSE
                DEST[7:0]←TEMP[31:24]
                DEST[15:8]←TEMP[23:16]
                DEST[23:16]←TEMP[15:8]
                DEST[31:24]←TEMP[7:0]
    ENDIF;
mov eax,12345678h
bswap eax    ;eax=78563412h
```

увеличим третий байт регистра EAX на единицу (не сам регистр)

```
mov eax,87654321h
bswap eax;eax=21436587h
inc al
bswap eax;eax=88654321h
```

Использование BSWAP с 16-разрядными регистрами дает неожиданные результаты

```
mov eax,0AB120000h
bswap ax;eax = 0AB1212ABh
```

## 7.5. Оператор указания типа (PTR)

По команде MOV можно переслать как байт и слово, так и двойное слово. А как узнать, что именно – байт или слово – пересылает команда? Не может ли получиться так, что мы хотели переслать слово, а оказалось, что команда пересылает байт? Размер пересылаемой величины обычно определяется по типу операндов, указанных в команде MOV.

*Пример:*

```
X DB ?       ;TYPE X = BYTE
Y DW ?       ;TYPE Y = WORD
MOV BH,0     ;пересылка байта
MOV X,0;пересылка байта (X описан как имя байтовой переменной)
MOV ESI,0    ;пересылка двойного слова
MOV Y,0 ;пересылка слова (Y описан как имя переменной-слова)
```

Обратите внимание, что по второму операнду (0) нельзя определить, какого он размера: ноль может быть и байтом (00h), и словом (0000h), и двойным словом (00000000h).

Если можно определить размеры обоих операторов, тогда эти размеры должны совпадать (либо байты, либо слова, либо двойные слова), иначе ассемблер зафиксирует ошибку.

*Пример:*

```
MOV DI,ES     ;пересылка слова
MOV CH,X      ;пересылка байта
```

```
MOV ESI,AX      ;ошибка (ESI регистр размером в двойное
;слово, AX регистр размером в слово)
MOV BH,300;ошибка – BH байтовый регистр, а 300 не может быть байтом
```

Оператор переопределения типа PTR (от **P**OIN**T**E**R**, указатель) применяется для переопределения или уточнения типа метки или переменной, определяемой выражением. Оператор переопределения типа PTR записывается следующим образом:

<*тип*> PTR <*выражение*>

Тип может принимать одно из следующих значений: BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR, а выражение может быть константным или адресным. Оператор PTR используется с атрибутами BYTE, WORD, DWORD и т.д. для локальной отмены определенных типов (DB, DW и т.д.) или с атрибутами NEAR, FAR для отмены значения дистанции по умолчанию.

Если указано константное выражение, то оператор говорит о том, что значение этого выражения (число) должно рассматриваться языком ассемблера как величина указанного типа (размера); например, BYTE PTR 0 – это ноль как байт, а WORD PTR 0 – это ноль как слово.

Если же указано адресное выражение, то оператор показывает, что адрес, являющийся значением выражения, должен восприниматься языком ассемблера как адрес ячейки указанного типа (размера); например, WORD PTR A – адрес A обозначает слово (байты с адресами A и A+1). В данном случае оператор PTR относится к адресным выражениям.

## Контрольные вопросы и упражнения

1. Верны ли приведённые ниже команды:

| | |
|---|---|
| а) mov ax, –4; | е) mov ax,bl; |
| б) mov ax,1000h; | ж) mov dl,ax; |
| в) mov [1000h],–16; | з) mov ax,ax; |
| г) mov ds,ax; | и) mov bl,al; |
| д) mov ds,1000h; | к) mov ip,cx? |

2. Пусть базовая единица информации компьютера содержит 12 бит, а в командах 3 бита задают режим каждого адреса, 4 бита – адрес каждого регистра, 24 бита – каждый адрес памяти и 8 бит – каждый адрес ввода/вывода. Определите:

а) количество режимов адресации;

б) число регистров;

в) емкость и адресное пространство памяти;

г) число адресов ввода/вывода;

д) диапазон целых чисел с одинарной точностью в дополнительном коде;
е) диапазон целых чисел с двойной точностью;
ж) диапазон двоично-десятичных упакованных чисел;
з) примерный диапазон вещественных чисел с плавающей запятой, если для порядка отведено 12 бит и он представлен в двоичном дополнительном коде;
и) примерную точность в значащих разрядах, если для мантиссы отведено 24 бита.

# ГЛАВА 8
# БУЛЕВА АЛГЕБРА

Характерная черта современных компьютеров – сведение всех вычислительных структур к двоичным числам и алгоритмам их обработки.

## Алгебра логики

Алгебра логики – раздел математической логики, в котором изучаются логические операции над высказываниями. Высказывания могут быть истинными (*true*) и ложными (*false*).

## Определение

Высказывания строятся над множеством $\{\Re, \neg, \wedge, \vee, 0, 1\}$, где $\Re$ – непустое множество, над элементами которого определены три базовых операции:

$\neg$ одноместная операция отрицание (*логическое НЕ*). Для обозначения операции *НЕ* в алгебре логики так же может использоваться черта над символом; «НЕ *A*» записывается как $\overline{A}$ ;

$\wedge$ двуместная операция конъюнкция (*логическое И*). Кроме знака $\wedge$ для обозначения операции *логическое И* иногда используется точка (·) или слитное написание операторов; операция «*A* и *B*» записывается как $A \cdot B$, или просто $AB$;

$\vee$ двуместная операция дизъюнкция (*логическое включающее ИЛИ*). Кроме знака $\vee$ для обозначения операции *логическое включающее ИЛИ* иногда используется +; операция «*A* или *B*» записывается как $A+B$, или $A \vee B$;

а также две константы – логический ноль 0 (*false*) и логическая единица 1 (*true*).

Операции над булевскими значениями (логическая конъюкция, дизъюнкция и отрицание) приведены в таблице 8.1.

*Таблица 8.1.*

**Булевские операции**

| $A$ | $B$ | $A \vee B$ | $A \wedge B$ | $\neg A$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 |

*Дизъюнкт* – пропозициональная формула, являющаяся дизъюнкцией двух или более литералов (например, $A \vee \overline{B} \vee C$ ). *Конъюнкт* – пропозициональная формула, являющаяся конъюнкцией двух или более литералов (например, $A \wedge \overline{B} \wedge C$ ). Для обозначения эквивалентности логических выражений используется знак равенства «=».

## Аксиомы

1. $\overline{\overline{A}} = A \; A = \overline{\overline{A}}$ ;
2. $A \cdot \overline{A} = 0$ ;
3. $A+1=1$;
4. $A+A=A$, $A=A+A+A$;
5. $A+0=A$;
6. $A \cdot A=A$, $A=A \cdot A \cdot A$;
7. $A \cdot 0=0$;
8. $A \cdot 1=A$;
9. $\overline{A} = 0^A$ ;
10. $\overline{A} = 1-A$ ;
11. $A+\overline{A}=1$ .

## Логические операции

Простейшим и наиболее широко применяемым примером такой алгебраической системы является множество $\Re$, состоящее всего из двух элементов:

$$\Re = \{Ложь, Истина\}$$

Как правило, в математических выражениях **Ложь** отождествляется с логическим нулём, а **Истина** – с логической единицей, а операции отрицания (НЕ), конъюнкции (И) и дизъюнкции (ИЛИ) определяются в привычном нам понимании. В общем случае логические выражения являются функциями логических переменных $A$,$B$,$C$, каждая из которых может иметь значения 0 или 1. Если имеется $k$ логических переменных, то они образуют $2^k$ возможных логических наборов из 0 и 1. При $k$=1: $A$=0 и $A$=1; при $k$=2: $AB$=00, 01, 10, 11 и так далее. Для каждого набора переменных логическая функция $F$ может принимать значение 0 или 1. Поэтому для $k$ переменных

можно образовать $l_k=2^{2^k}$ различных логических функций. Таким образом, при $k$=2 можно получить $l_2$=16 функций и далее при увеличении $k$ число $l_k$ растет чрезвычайно быстро: $l_3$=256, $l_4$=65536 и так далее. На данном множестве $\Re$ с $k$=2 можно задать шесть одноместных ( $A, B, \overline{A}, \overline{B}, 0, 1$ ) и десять двуместных функций ( $A \vee B, A \wedge B, A \to B, A \downarrow B, B \to A$, $A \leftrightarrow B$, $A \oplus B$ и другие), представленных в таблице 1, однако все функции могут быть получены через суперпозицию трех операций И, ИЛИ, НЕ.

*Таблица 8.2*

**Полный набор логических функций для переменных A и B**

| переменные | | функции | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | $B$ | 0 | $A \wedge B$ | $\overline{A \to B}$ | $A$ | $\overline{B \to A}$ | $B$ | $A \oplus B$ | $A \vee B$ | $A \downarrow B$ | $A \leftrightarrow B$ | $\overline{B}$ | $B \to A$ | $\overline{A}$ | $A \to B$ | $A \mid B$ | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | | 0 | $A\,B$ | $A\overline{B}$ | $A$ | $\overline{A}B$ | $B$ | $\overline{A}B+A\overline{B}$ | $A+B$ | $\overline{A+B}$ | $AB+\overline{A}\overline{B}$ | $\overline{B}$ | $A+\overline{B}$ | $\overline{A}$ | $\overline{A}+B$ | $\overline{AB}$ | 1 |
| | | суперпозиция | | | | | | | | | | | | | | | |

Опираясь на этот математический инструментарий, логика высказываний изучает высказывания и предикаты. Также вводятся дополнительные операции, такие как эквивалентность $\leftrightarrow$ («тогда и только тогда, когда»), импликация $\to$ («следовательно»), сложение по модулю два $\oplus$(«исключающее или»), штрих Шеффера $\mid$, стрелка Пирса $\downarrow$ и другие.

Все указанные операции, за исключением одноместных (НЕ и Тождественность), выполняются при двух и более переменных на входе, и в каждом случае получается только один выход. Одноместные операции характеризуются одним входом и одним выходом.

Логика высказываний послужила основным математическим инструментом при создании компьютеров. Она легко преобразуется в битовую логику: истинность высказывания обозначается одним битом (0 – ЛОЖЬ, 1 – ИСТИНА); тогда операция $\neg$ приобретает смысл вычитания из единицы; $\vee$– немодульного сложения; $\wedge$ – умножения; $\leftrightarrow$ – равенства; $\oplus$ – сложение по модулю 2 (исключающее ИЛИ – XOR); $\mid$ – непревосходства суммы над 1 (то есть $A \mid B = (A + B) \le 1$) и так далее.

Впоследствии булева алгебра была обобщена от логики высказываний путём введения характерных для логики высказываний аксиом.

## Свойства логических операций

Коммутативность (переместительность аргументов, «от перестановки аргументов результат не изменяется»):

$$A\#B = A\#B,\ \# \in \{\vee,\wedge,\oplus,\leftrightarrow,\downarrow,|\}.$$

Идемпотентность:

$$A\#A = A,\ A = A\#A,\ \#\in \{\vee,\wedge\}.$$

Ассоциативность (соединительность аргументов):

$$(A\#B)\#C = A\#(B\#C),\ \#\in \{\vee,\wedge,\oplus,\leftrightarrow\}.$$

Дистрибутивность конъюнкции и дизъюнкции относительно дизъюнкции, конъюнкции и суммы по модулю два соответственно:

$$A\cdot(B+C)=(A\cdot B)+(A\cdot C),$$
$$A+(B\cdot C)=(A+B)\cdot(A+C),$$
$$A\cdot(B\oplus C)=(A\cdot B)\oplus(A\cdot C)$$

Законы де Моргана:

$$\overline{A\,B}=\overline{A}+\overline{B}$$
$$\overline{A+B}=\overline{A}\ \overline{B}$$

Законы поглощения:

$$A\cdot(A+B) = A,$$
$$A+(A\cdot B) = A.$$

Другие:

1. $A\cdot\overline{A}=A\cdot 0=A\ \oplus A = 0.$
2. $A+\overline{A}=A+1=A\ \leftrightarrow A = A\rightarrow A = 1.$
3. $A+A=A\cdot A=A\cdot 1=A+0=A\ \oplus 0 = A.$
4. $A\oplus 1 = A\rightarrow 0 = A\leftrightarrow 0 = A\,|\,A = A\downarrow A = \overline{A}$ .
5. $\overline{\overline{A}}=A$
6. $A\oplus B=(A\cdot\overline{B})+(\overline{A}\cdot B)=(A+B)\cdot(\overline{A}\vee\overline{B})$.
7. $A\leftrightarrow B=\overline{x\oplus y}=(A\cdot B)+(\overline{A}\cdot\overline{B})=(A+\overline{B})\cdot(\overline{A}\vee B)$ .
8. $A\rightarrow B=\overline{A}+B=((A\cdot B)\oplus A)\oplus 1.$

Дополнение законов де Моргана:

9. $A\,|\,B=\neg(x\wedge y)=\overline{A}+\overline{B}$ .
10. $A\downarrow B=\neg(x\vee y)=\overline{A}\cdot\overline{B}$ .

Существуют методы упрощения логической функции: например, Карта Карно, метод Квайна – Мак-Класки.

## Законы де Моргана

*Формулировка*: Если существует операция логического умножения двух и более элементов $A \cdot B$, то для того чтобы найти обратное от всего суждения $\overline{AB}$ необходимо найти обратное от каждого элемента и объединить их операцией логического сложения.

Два закона булевой алгебры, позволяющие представить дополнение сложного выражения через дополнения его членов в виде

$$\overline{A\,B} = \bar{A} + \bar{B}$$

$$\overline{A + B} = \bar{A}\;\bar{B}$$

Закон работает и в обратном направлении.

Термин «законы де Моргана» часто употребляется применительно к аналогичным соотношениям, используемым в других случаях, например, при операциях над множествами или логическими выражениями.

## Булева алгебра

Булевой алгеброй называется непустое множество $\Re$ с двумя двуместными операциями $\wedge$ (аналог конъюнкции), $\vee$ (аналог дизъюнкции), одноместной операцией $\neg$ (аналог отрицания) и двумя выделенными элементами: 0 (или «Ложь») и 1 (или «Истина») такими, что для всех $A$, $B$ и $C$ из множества $\Re$ верны следующие аксиомы:

| | | |
|---|---|---|
| $A+B+C=(A+B)+C=A+(B+C)$ | $ABC=(AB)C=A(BC)$ | ассоциативность |
| $A+B=B+A$ | $AB=BA$ | коммутативность |
| $A+AB=A$ | $A(A+B)=A$ | законы поглощения |
| $A(B+C)=AB+AC$ | $A+BC=(A+B)(A+C)$ | дистрибутивность |
| $A+\overline{A}=1$ | $A\cdot\overline{A}=0$ | дополнительность |

Первые три аксиомы означают, что ($\Re$, $\wedge$, $\vee$) является решёткой. Таким образом, булева алгебра может быть определена как дистрибутивная решётка, в которой выполнены две последние аксиомы. Структура, в которой выполняются все аксиомы, кроме предпоследней, называется *псевдобулевой алгеброй*.

Булева алгебра имеет практическое приложение в цифровой технике, основанной на двоичной логике. Как существуют булевы функции, так существуют и булевы производные. Булевы производные – единственный математический аппарат для разработки тестов цифровой техники.

## Абстрактное определение булевой алгебры

Множество элементов с заданными на них двуместными операциями «логическое И» и «логическое ИЛИ» (конъюнкция и дизъюнкция), удовлетворяющими законам коммутативности, ассоциативности, идемпотентности и поглощения, называется структурой, а если выполняется еще и закон дистрибутивности, то и дистрибутивной структурой. В случае, когда к указанным выше операциям добавляется еще одна одноместная инволютивная операция НЕ (отрицание), причем удовлетворяются законы де Моргана и законы нейтральности, говорят о булевой структуре или булевой алгебре.

**Основные аксиомы и законы булевой алгебры**

| | |
|---|---|
| аксиомы | $\overline{0}=1$<br>$\overline{1}=0$<br>$A\cdot 0=0$<br>$A+1=1$<br>$A+0=A$<br>$A\cdot 1=A$<br>$\overline{A}\cdot A=0$<br>$\overline{A}+A=1$ |
| Закон инволюции | $\overline{\overline{A}}=A$ |
| Закон коммутативности | $A+B=B+A$<br>$AB=BA$ |
| Закон ассоциативности | $A+B+C=(A+B)+C=A+(B+C)$<br>$ABC=(AB)C=A(BC)$ |
| Закон дистрибутивности | $A(B+C)=AB+AC$<br>$A+BC=(A+B)(A+C)$ |
| Законы дуальности (де Моргана) | $\overline{AB}=\overline{A}+\overline{B}$<br>$\overline{A+B}=\overline{A}\,\overline{B}$ |
| Закон поглощения | $A+AB=A$<br>$A(A+B)=A$ |
| Закон нейтральности | $A+\overline{B}\cdot B=A$<br>$A\cdot(\overline{B}+B)=A$ |

| | |
|---|---|
| Закон идемпотентности | $A\cdot A=A$<br>$A+A=A$ |

$A \le B$ тогда и только тогда, когда $A\cdot B=A$.

Используя данные тождества и законы, можно получать новые логические выражения, а также доказывать справедливость тех или иных законов.

Например, с помощью закона дистрибутивности $A+BC=(A+B)(A+C)$ и тождества $\overline{A}+A=1$ получаем соотношение Блейка-Порецкого:

$$A+\overline{A}B=(A+\overline{A})(A+B)=A+B$$

Используя закон дистрибутивности $A(B+C)=AB+AC$ тождества $A+1=1$ и $A\cdot A=A$ закон ассоциативности $A\cdot B\cdot C=(A\cdot B)\cdot C=A\cdot(B\cdot C)$, получаем доказательство закона поглощения:

$$A\cdot(A+B)=A\cdot A+A\cdot B=A+A\cdot B=A\cdot(1+B)=A$$

Склеивания

$$(A+B)(\overline{A}+B)=A\overline{A}+AB+B\overline{A}+BB=0+AB+B\overline{A}+B=B(A+\overline{A})+B=B+B=B$$

Закон поглощения:

$$A+AB=A(1+B)=A\cdot 1=A$$

Закон дистрибутивности:

$$(A+B)(A+C)=AA+AC+BA+BC=A+AC+BA+BC=A(1+C)+BA+BC=A\cdot 1+BA+BC=A(1+B)+$$
$$+BC=A\cdot 1+BC=A+BC$$

Закон дистрибутивности:

$$A+BC=A\cdot 1+BC=A(1+B)+BC=A\cdot 1+BA+BC=$$
$$=A(1+C)+BA+BC=A+AC+BA+BC=AA+AC+BA+BC=A(A+C)+B(A+C)=(A+B)(A+C)$$
$$A+\overline{A}B=(A+\overline{A})(A+B)=A+B$$

Используя логические соотношения $\overline{A}\cdot A=\overline{B}\cdot B=0$, можно получить

$$A\oplus B=\overline{A}\cdot A+A\cdot\overline{B}+B\cdot\overline{A}+B\overline{B}=A\cdot(\overline{A}+\overline{B})+B\cdot(\overline{A}+\overline{B})=$$
$$=A\cdot(\overline{AB})+B\cdot(\overline{AB})=(A+B)\overline{AB}$$

## Принцип двойственности

В булевых алгебрах существуют двойственные утверждения, они либо одновременно верны, либо одновременно неверны. Если в формуле, которая верна в некоторой булевой алгебре, поменять все конъюнкции на дизъюнкции, 0 на 1, $\le$ на $\ge$ и наоборот, то получится формула, также истинная в этой булевой алгебре. Это следует из симметричности аксиом относительно таких замен.

## Булевы операции и математическая логика

Булевы операции очень близки (хотя и не тождественны) логическим связкам в классической логике. Бит можно рассматривать как логическое суждение – его значениями являются 1 «истина» и 0 «ложь». При такой интерпретации известные в логике связки конъюнкции, дизъюнкции, импликации, отрицания и другие имеют представление на языке битов. И наоборот, битовые операции легко описываются на языке исчисления высказываний.

Связкам математической логики более соответствуют логические операции в том числе в программировании, чем собственно битовые операции.

Высказывания могут быть *простыми* и *сложными*. Сложное высказывание образуется в результате объединения простых высказываний с помощью логических связей. Сложные высказывания, получаемые из простых, будут истинными или ложными в зависимости от истинности или ложности простых высказываний, входящих в сложные. Значение сложного высказывания, так же как и простого, может принимать значение 0 или 1. Сложное высказывание можно рассматривать как электрический сигнал на выходе некоторой преобразующей схемы. Значение его будет равно 0 или 1 в зависимости от значений сигналов (простых высказываний) на входе рассматриваемой схемы. Это позволяет применять символику алгебры логики для анализа и синтеза цифровых схем.

### Булевы операции как основа цифровой техники

Булевы операции лежат в основе обработки цифровых сигналов. Через булевы операции можно из одного или нескольких сигналов на входе получить на выходе новый сигнал, который, в свою очередь, может быть подан на вход одной или нескольким таким операциям. Булевы операции в сочетании с запоминающими элементами (например, триггерами) реализуют всё богатство возможностей современной цифровой техники.

## Основные булевы операции

### Операция НЕ (Инверсия)

«Логическое НЕ (NOT)», инвертирование – аналог отрицания в логике. Данная одноместная операция (с одним входом) заменяет 0 на 1 и наоборот. Реализующий её элемент называется *инвертором*. НЕ инвертирует любую двоичную цифру или группу цифр.

| ***Вход A*** | ***Выход*** $= \overline{A}$ | **Аксиомы, законы и следствия** |
|---|---|---|
| 0 | 1 | $\overline{0} = 1$ $\quad \overline{\overline{A}} = A$ |
| 1 | 0 | $\overline{1} = 0$ $\quad \overline{A} = 1 - A$ |

## Операция Логическое И

«Логическое И (AND)» – аналог конъюнкции в логике. Иногда называется логическим умножением. Выход вентиля И принимает значение 1 только в том случае, если обе входные переменные *A* и *B*, то на выходе появляется 1, во всех остальных случаях выход принимает значение 0. В общем случае число входов вентиля не ограничено.

| ***Входы*** | | ***Выход = A and B*** |
|---|---|---|
| ***A*** | ***B*** | |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$A \cdot A = A$

$A \cdot 1 = A$

$A \cdot 0 = A$

$A \overline{A} = 0$

$A \cdot B = B \cdot A$

$A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$

## Логическое ИЛИ

«Логическое ИЛИ (OR)» – аналог *дизъюнкции* в логике. Выход вентиля ИЛИ принимает значение 0 если обе входные переменные A и B принимают значение 0, во всех остальных случаях выход равен 1. В общем случае число входов вентиля не ограничено. Операция двойственна AND: при инвертировании выхода и всех входов (то есть при замене 0 и 1 местами) «И» и «ИЛИ» взаимно превращаются друг в друга.

| ***Входы*** | | ***Выход = A or B*** |
|---|---|---|
| ***A*** | ***B*** | |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$A + A = A$

$A + 1 = 1$

$A + 0 = A$

$\overline{A} + A = 1$

$A + B = B + A$

$A + B + C = (A + B) + C = A + (B + C)$

## Прочие битовые операции

Операции могут совмещать инвертирование с выполнением функций И и ИЛИ.

«ИЛИ-НЕ (NOR, «стрелка Пирса»)». Стрелка Пирса является результатом инвертирования результата «ИЛИ» своих аргументов, выдаёт значение 1 только когда оба входа 0.

«И-НЕ (NAND, штрих Шеффера)». Двойственная стрелке Пирса операция: является результатом инвертирования результата «И» своих аргументов, выдаёт значение 0 только когда оба входа 1.

*Импликация* («если-то») – аналог импликации в логике. Совпадает с «ИЛИ» с инвертированным первым аргументом, выдаёт значение 0 только когда первый вход 1 а второй – 0. Данная операция не является коммутативной, в отличие от всех вышеописанных бинарных операций. Её можно понимать как арифметическое $\leq$ (меньше или равно).

*Эквиваленция*. Выдаёт 1 если и только если оба аргумента равны между собой. Является результатом инвертирования результата «исключающего ИЛИ» своих аргументов. Она же и двойственна исключающему «ИЛИ» в вышеописанном смысле.

## Исключающее ИЛИ

«Исключающее ИЛИ (XOR, сложение по модулю 2)» – аналог исключающего ИЛИ в логике. Если входные переменные $A$ и $B$ имеют разное значение, то выход принимает значение 1, во всех остальных случаях выход равен 0. Операция обозначается символами $\oplus$ или $\veebar$. Результат операции XOR отличается от результата OR только в случае одновременного равенства аргументов 1: $A \oplus B = A + B - AB$. Значение данной логической связки можно передавать союзом «либо».

Результат операции XOR является сложением в кольце вычетов по модулю 2, в отличие от операций «И» и «ИЛИ» данная операция является обратимой, или инволютивной: $(A \oplus B) \oplus B = A$.

| ***Входы*** | | ***Выход = A xor B*** |
|---|---|---|
| ***A*** | ***B*** | |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

$$A \oplus 0 = A$$

$$A \oplus A = 0$$

$$A \oplus \overline{A} = 1$$

$$A \oplus 1 = \overline{A}$$

$$A \oplus B = B \oplus A$$

$$A \oplus B \oplus C = (A \oplus B) \oplus C = A \oplus (B \oplus C)$$

$$(A \oplus B) \oplus B = A$$

В компьютерной графике «исключающее ИЛИ» применяется в компьютерной анимации при выводе элементов изображения (*спрайтов*) на экран – повторное применение операции XOR к тем же элементам изображения стирает спрайты с экрана. Благодаря инволютивности операция XOR нашла применение в криптографии как простейшая реализация шифра Вернама. «Исключающее ИЛИ» также может использоваться для обмена двух переменных, используя алгоритм обмена при помощи исключающего ИЛИ.

## Операции от многих аргументов

Операции «И», «ИЛИ» и «исключающее ИЛИ» являются не только коммутативными, но и ассоциативными, и потому легко обобщаются на случай нескольких аргументов (входов).

## Операции над битовыми векторами

### Обобщение операций на булеву алгебру

Теперь вместо одиночных битов рассмотрим векторы из фиксированного количества битов (для нас это регистры и ячейки памяти). Регистр можно рассматривать, как двоичное разложение целого числа:

$$b = b_0 + 2b_1 + 2^2 b_2 + \ldots + 2^{N-1} b_{N-1}$$

($N$ – количество битов в регистре) и булевые операции проводят покомпонентно (значение в $k$-ом бите – результат операции над $k$-ми битами аргументов). Булевы операции распространяются таким образом на произвольную булеву алгебру. Таким образом мы получаем операции побитового AND, OR, NOT, XOR и так далее. Побитовое NOT для чисел в дополнительном коде совпадает с вычитанием из минус единицы.

$$\overline{A} = -1 - A$$
$$-A = \overline{A} + 1$$
$$2^N - 1 = 1111\ldots11 \text{ (N раз)}$$
$$A \text{ and } (-1) = A$$
$$A \oplus (-1) = \overline{A}$$
$$A \text{ or } (-1) = -1$$
$$A \oplus \overline{A} = -1$$

### Физическая реализация битовых операций

Реализация логических и битовых операций может в принципе быть любой: механической, электромеханической, гидравлической, пневматической, оптической и даже химической. В первой половине XX века до изобретения транзисторов в вычислительной технике применяли электромеханические реле и электронные лампы. С 50-х годов XX века распространяются электронные реализации логических и битовых операций при помощи транзисторов.

### Использование в программировании

Благодаря аппаратной реализации в арифметическом логическом устройстве (АЛУ) процессора многие регистровые битовые операции аппаратно доступны в языке ассемблера. В большинстве процессоров реализованы в качестве инструкции регистровый НЕ; регистровые двухаргумент-

ные И, ИЛИ, исключающее ИЛИ; проверка равенства нулю; три типа битовых сдвигов, а также циклические битовые сдвиги.

Регистровая операция И используется для сброса конкретных битов по битовой маске, ИЛИ – для установки, исключающее ИЛИ – для инвертирования битов регистра по маске.

## Логические соотношения

Теоретической основой проектирования цифровых систем является алгебра логики или булева алгебра. В алгебре логики различные выражения (высказывания) могут иметь только два значения — «истинно» или «ложно». Для обозначения истинности или ложности высказываний пользуются символами 1 или 0. Это хорошо согласуется с двоичной системой счисления и с работой двухпозиционных элементов, используемых в микропроцессоре. Например, истинность высказывания может быть представлена в микропроцессоре сигналом положительного (отрицательного) электрического напряжения, а ложность – отрицательным (положительным) напряжением или отсутствием сигнала вообще.

## Однобитная логика

В таблице дано табличное представление логических операций отрицания, сложения, умножения, исключающего ИЛИ для переменных A, B и их дополнениями $\overline{A}$ , $\overline{B}$ , а также результат инверсии логического сложения и умножения.

| $A$ | $B$ | $\overline{A}$ | $\overline{B}$ | $A\,B$ | $A+B$ | $\overline{A}\,\overline{B}$ | $\overline{A\,B}$ | $\overline{A}+\overline{B}$ | $\overline{A+B}$ | $A\oplus B$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |

## Контрольные вопросы и упражнения

1. Чему равны следующие соотношения а) 0·1 б) 0+1 в) 1·1 г) 1+1 д) A(A+B) е) A( $\overline{A}$ +B) ж) A⊕A з) A⊕ $\overline{A}$ и) A(A+A·B) ?
2. Как модифицировать операцию «Исключающее ИЛИ» что бы получилась инверсия?
3. С помощью логических преобразований покажите, что

$A \oplus B = \overline{AB + \overline{B}\,\overline{A}}$

$A \oplus B = (A + B)(\overline{A} + \overline{B})$

4. С помощью логических преобразований выведите уравнение Хантингтона

$$\overline{(\overline{X} + Y)} + \overline{(\overline{X} + \overline{Y})} = X$$

5. С помощью логических преобразований выведите уравнение Роббинса

$$\overline{\overline{(X + Y)} + (X + \overline{Y})} = X$$

6. Используя взаимные замены операций AND и OR и символов 0 и 1 расположите в одной таблице тождества и законы алгебры логики.

| AND | OR | XOR |
|---|---|---|
| | | |

ГЛАВА 9

# ЛОГИЧЕСКИЕ КОМАНДЫ

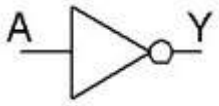
Булева алгебра располагает тремя основными операциями – И, ИЛИ, НЕ, которые позволяют производить сложение, вычитание, умножение, деление и сравнение символов и чисел. Любые функции, которые Вы только пожелаете реализовать, можно осуществить, комбинируя вентили названных типов. Для всех функций, реализуемых в компьютере, имеется свое схемное представление, которое вступает в действие каждый раз, когда в регистр команд попадает соответствующая команда.

Логические команды выполняют логические операции, известные вам по языкам высокого уровня – инверсию, конъюнкцию и дизъюнкцию. Логические команды реализуют поразрядные операции: *i*-й разряд результата зависит только от *i*-х разрядов операндов. Логическая операция выполняется сразу над всеми разрядами операндов одновременно, параллельно.

Операндами логических команд должны быть либо байты, либо слова, либо двойные слова, либо счетверенные слова, но не то и другое одновременно (размер операндов должен *обязательно* совпадать!). В качестве второго операнда могут также использовать так называемую маску, записанную не в шестнадцатеричном, а в двоичном коде.

## 9.1.1. Команда NOT

(Логическое отрицание = **logical NOT**)




| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| NOT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Синтаксис команды:* NOT < *DEST*>

*Возможные варианты команды:*

```
not reg/mem
```

*Семантика команды:* операция логического отрицания для операнда *DEST* размерностью байт, слово или двойное слово.

*Псевдокод:*

*DEST←NOT DEST*

*Алгоритм работы:*

- изменить значение каждого бит операнда *DEST* на противоположное: 0 на 1, 1 на 0;
- записать результат операции в операнд *DEST*;
- установить флаг нуля ZF, если получился нулевой результат.

*Применение:* Команда NOT используется для работы с операндами на уровне битов.

```
MOV AL,0F3h;AL=11110011b
NOT AL ;AL=0Ch=00001100b
```

Так как для получения числа в дополнительном коде используется сперва инвертирование значения, а потом добавляется 1, то команда `NOT DEST` может заменить последовательности команд `NEG DEST`/`DEC DEST` или `IMUL DEST,-1`/`DEC DEST` или `XOR DEST,-1`

А вот так, если «размазать» по коду последовательности команд, можно «спрятать» команду `NOT AL`

**1-й вариант**

```
MOV AL,0F3h
NEG AL;AL=0Dh
DEC AL;AL=0Ch
```

**2-й вариант**

```
MOV AL,0F3h
IMUL EAX,-1;AL=0Dh
DEC AL;AL=0Ch
```

**3-й вариант**

```
XOR BL,BL;BL=0
SUB BL,AL;BL=0Dh
DEC BL;BL=NOT AL
```

**4-й вариант**

```
MOV AL,0F3h
XOR AL,-1;AL=0Ch
```

**5-й вариант**

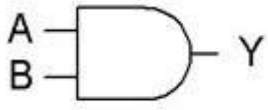
```
OR BL,-1
SUB BL,AL;BL=NOT AL
```

**6-й вариант**

```
INC AL;AL=0F4h
NEG AL;AL=0Ch
```

### 9.1.2. Команда AND

(Логическое И = **logical AND**)




| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| AND | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ANDPD | | | | | | | ✓ | ✓ | ✓ | ✓ |
| ANDPS | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| PAND | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Синтаксис команды:*

AND <*DEST*>,<*SRC*>

*Возможные варианты команды:*

```
        and reg/mem,reg(8/16/32/64)
        and reg,mem(8/16/32/64)
        and reg/mem,imm(8/16/32)
MMX     pand mmx,m64/mmx
SSE     andps xmm,m128/xmm
SSE2    andpd xmm,m128/xmm
        pand xmm,m128/xmm
```

*Семантика команды:* операция логического умножения над битами *операнда DEST* размерностью байт, слово, двойное, учетверенное слово или параграф.

*Псевдокод:*

$$DEST \leftarrow DEST \wedge SRC$$

*Алгоритм работы:*

- выполнить операцию логического умножения над битами *операнда DEST*, используя операнд *SRC*. При этом бит результата равен 1, если соответствующие биты *DEST* и *SRC* равны 1, в противном случае бит равен 0;
- записать результат операции в операнд *DEST* (операнд *SRC* остается неизменным);
- установить флаги.

*Применение: к*оманда AND используется для работы с операндами на уровне битов. Удобно использовать для принудительного сброса определенных битов операнда.

```
and bl,11110110b     ;сбросить нулевой и третий биты регистра BL
    mov eax,00000a5a5h
    mov edx,000000ff0h
    and eax,edx
```

| переменная | Bin | Hex |
|---|---|---|
| eax | 00000000000000001010010110100101b | 0A5A5h |
| edx | 00000000000000000000111111110000b | 00FF0h |
| eax=eax ∧edx | 00000000000000000000010110100000b | 005A0h |

```
msk   dd       00000FFFFFFFF0000h
dat   dd       00000A5A50000FF11h
mov   esi,offset dat   ; Data pointer
mov   ebx,offset msk   ; Mask pointer
movq mm7,[esi]        ; Get 64 bits of data
pand mm7,[ebx]        ; AND it with the mask data
movq [edi],mm7        ; Save the masked data
```

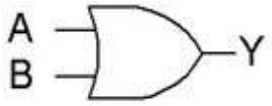
Можно использовать эту команду для сравнения значения регистра с нулевым или ненулевым значением. Например, надо проверить, равен ли регистр EAX нулю. Для этого можно было бы применить команду `cmp eax,0`, но вместо этого используют `and eax,eax` и после этой инструкции помещают условный переход `JZ` или `JNZ`. Логическая операция AND, выполненная по отношению к одному и тому же числу, дает в результате это же число. Инструкция AND, как и любая другая математическая инструкция, устанавливает флаги, включая флаг нуля. Результат выполнения AND будет равен нулю только в том случае, если число в регистре нулевое.

Команда `cmp eax,0` занимает 3/5 байт и выполняется за 4 такта синхронизации, команда `and eax,eax` занимает 2 байта и выполняется на 1 такт быстрее. Команду AND можно использовать вместо MOV для размещения в ячейке памяти/регистре нулевого значения. В отличие от MOV логические команды могут использовать знаковое расширение числа.

| команда | кодировка | длина | комментарий |
|---|---|---|---|
| `mov eax,0` | `0B800000000h` | `5 байт` | `Без знакового расширения` |
| `and eax,0` | `2500000000h` | `5 байт` | `Без знакового расширения` |
| `and eax,0` | `83E000h` | `3 байта` | `Со знаковым расширением` |

### 9.1.3. Команда OR

(Логическое включающее ИЛИ = **logical OR**)




| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| OR | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ORPD | | | | | | | ✓ | ✓ | ✓ | ✓ |
| ORPS | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| POR | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Синтаксис команды:*

OR *<DEST>,<SRC>*

*Возможные варианты команды:*

```
        or reg/mem,reg(8/16/32/64)
        or reg,mem(8/16/32/64)
        or reg/mem,imm(8/16/32)
MMX     por mmx,m64/mmx
```

```
SSE     orps xmm,m128/xmm
SSE2    orpd xmm,m128/xmm
        por xmm,m128/xmm
```

*Семантика команды:* операция логического ИЛИ над битами операнда *DEST*.
*Псевдокод:*

$$DEST \leftarrow DEST \vee SRC$$

*Алгоритм работы:*

- выполнить операцию логического ИЛИ над битами операнда *DEST*, используя операнд *SRC*. При этом бит результата равен 0, если соответствующие биты операнда *DEST* и *SRC* равны 0, в противном случае бит результата равен 1;
- записать результат операции в операнд *DEST* (операнд *SRC* остается неизменным);
- установить флаги.

*Применение:* команду OR можно использовать для работы с операндами на уровне битов. Типичное использование команды – установка определенных разрядов операнда *DEST* в единицу.

```
OR BL,00000001b     ;установить нулевой бит регистра BL в 1
    mov eax,00000a5a5h
    mov edx,000000ff0h
    or eax,edx
```

| переменная | Bin | Hex |
|---|---|---|
| eax | 00000000000000001010010110100101b | 0A5A5h |
| edx | 00000000000000000000111111110000b | 00FF0h |
| eax=eax∨edx | 00000000000000000000010110100000b | 0AFF5h |

```
msk   dd       00000ffffffff0000h
dat   dd       00000a5a50000ff11h
mov   esi,offset dat ; Data pointer
mov   ebx,offset msk ; Mask pointer
movq mm7,[esi]       ; Get 64 bits of data
por mm7,[ebx]        ; OR it with the mask data
movq [edi],mm7       ; Save the masked data
```

Эту команду можно использовать для сравнения значения регистра с нулевым или ненулевым значением. Например, надо проверить регистр AX, чтобы выяснить равен ли он нулю. Можно было бы применить команду `cmp ax,0`, но используют `or ax,ax` и после этой команды помещают условный переход `jz` или `jnz`. Логическая операция OR, выполненная по отношению к одному и тому же числу, дает в результате

это же число. Инструкция OR, как и любая другая математическая инструкция, устанавливает флаги, включая флаг нуля. Но результат выполнения OR будет равен нулю только в том случае, если число в регистре равно нулю.

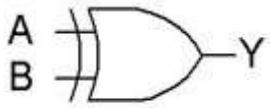
Команда `cmp eax,0` занимает 3/5 байта и выполняется за 4 такта синхронизации, команда `or eax,eax` занимает 2 байта и выполняется на 1 такт быстрее.

| команда | кодировка |
|---|---|
| CMP EAX,0 | 3D00000000 |
| CMP EAX,0 | 83F800 |
| OR EAX,EAX | 0BC0 |

Запись `or eax,8000h` эквивалентна `or ah,80h`

### 9.1.4. Команда XOR

(Логическое исключающее ИЛИ = **logical EXCLUSIVE OR**)




| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| XOR | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ORPD | | | | | | | ✓ | ✓ | ✓ | ✓ |
| XORPS | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| PXOR | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Синтаксис команды:*

XOR <*DEST*>,<*SRC*>

*Возможные варианты команды:*

```
        xor reg/mem, reg
        xor reg, mem
        xor reg/mem, imm
MMX     pxor mmx,m64/mmx
SSE     xorps xmm,m128/xmm
SSE2    xorpd xmm,m128/xmm
        pxor xmm,m128/xmm
```

*Семантика команды:* операция логического исключающего ИЛИ над операндом *DEST* размерностью байт, слово или двойное слово.

*Алгоритм работы:*

- выполнить операцию логического исключающего ИЛИ над битами *операнда DEST*, используя операнд *SRC*. При этом бит результата равен 1, если соответствующие биты операнда *DEST* и *SRC* различны, в остальных случаях бит результата равен 0;
- записать результат операции в операнд *DEST* (операнд *SRC* остается неизменным);
- установить флаги.

*Псевдокод:*

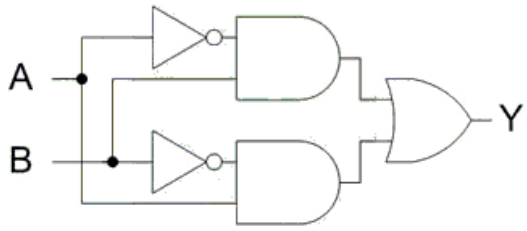
$$DEST \leftarrow DEST \oplus SRC$$

*Применение:* эту операцию удобно использовать для инвертирования или сравнения определенных битов операндов. Используется для простейшего шифрования и дешифровки:

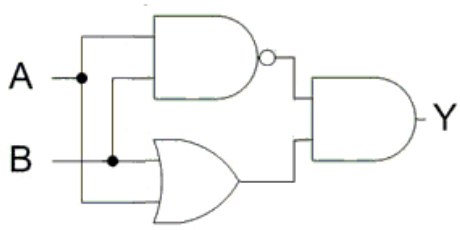
```
;изменить значение бита 0 регистра AL на обратное
 XOR AL,00000001b
    mov eax,00000a5a5h
    mov edx,000000ff0h
    or eax,edx
```

| переменная | Bin | Hex |
|---|---|---|
| eax | 00000000000000001010010110100101b | 0A5A5h |
| edx | 00000000000000000000111111110000b | 00FF0h |
| eax=eax∨edx | 00000000000000001010101001010101b | 0AA55h |

Операцию XOR можно создать комбинируя AND, OR и NOT.




| A | B | A' | B' | A'^B | A^B' | A'^B V A^B' |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | **0** |
| 0 | 1 | 1 | 0 | 1 | 0 | **1** |
| 1 | 0 | 0 | 1 | 0 | 1 | **1** |
| 1 | 1 | 0 | 0 | 0 | 0 | **0** |




| A | B | A∨B | (A^B)' | A∨B∧(A^B)' |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | **0** |
| 0 | 1 | 1 | 1 | **1** |
| 1 | 0 | 1 | 1 | **1** |
| 1 | 1 | 1 | 0 | **0** |

Но наиболее часто используют эту команду для обнуления значения регистра. Результат выполнения XOR над двумя битами будет равен единице только в том случае, когда один бит установлен в ноль, а другой в единицу. Логическая операция XOR, выполненная по отношению к

одному и тому же числу, даст в результате ноль. Команда `mov eax,0` занимает 5 байт и выполняется за 4 такта синхронизации, а команда `xor ax,ax` – 2 байта и выполняется на 1 такт быстрее.

### Шифрование с помощью команды XOR

*Шифр простой замены*. Возьмем символы ‘A’, ‘B’, ‘C’ и выполним операцию XOR с символом ‘1’. В зашифрованном тексте на месте символов A, B и C будут стоять ‘p’, ‘s’ и ‘r’ соответственно.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 01000001 | ;код «A» | | 01000010 | ;код «B» | | 01000011 | ;код «C» |
| ⊕ | 00110001 | ;код «1» | ⊕ | 00110001 | ;код «1» | ⊕ | 00110001 | ;код «1» |
| | 01110000 | ;код «p» | | 01110011 | ;код «s» | | 01110010 | ;код «r» |

Для дешифровки полученного таким способом текста этот текст посимвольно выполняет операцию XOR с тем же ключом, которым этот текст шифровался.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 01110000 | ;код «p» | | 01110011 | ;код «s» | | 01110010 | ;код «r» |
| ⊕ | 00110001 | ;код «1» | ⊕ | 00110001 | ;код «1» | ⊕ | 00110001 | ;код «1» |
| | 01000001 | ;код «A» | | 01000010 | ;код «B» | | 01000011 | ;код «C» |

Для дешифровки текста такого рода необходимо знать, на каком языке он написан. Приблизительная частота распределения букв уже давно подсчитана для всех языков мира. Допустим, вы знаете что зашифрованный текст на русском языке. Вы подсчитываете сколько всего символов в дешифруемом тексте. Затем подсчитывается, сколько раз каждый символ встречается в шифротексте. Полученное число делится на общее количество символов в тексте. Так вы получаете частоту употребления символов в вашей шифровке. Затем заменяете символы в зашифрованном тексте на буквы с аналогичными или близкими частотами. То есть, максимально часто встречающимся символом будет скорее всего пробел, на втором месте по частоте идет буква ‘o’, за ней ‘e’ и так далее...

*Таблица 9.1.1*

**Частота распределения букв в русском языке**

| буква | частота | буква | частота | буква | частота | буква | частота | буква | частота |
|---|---|---|---|---|---|---|---|---|---|
| а | 0,062 | з | 0,016 | о | 0,090 | х | 0,01 | ь | 0,014 |
| б | 0,014 | и | 0,062 | п | 0,023 | ц | 0,004 | э | 0,003 |
| в | 0,038 | й | 0,010 | р | 0,040 | ч | 0,012 | ю | 0,006 |
| г | 0,013 | к | 0,028 | с | 0,045 | ш | 0,006 | я | 0,018 |
| д | 0,025 | л | 0,035 | т | 0,053 | щ | 0,003 | пробел | 0,174 |
| е, ё | 0,072 | м | 0,026 | у | 0,021 | ъ | 0,014 | | |
| ж | 0,007 | н | 0,053 | ф | 0,002 | ы | 0,016 | | |

Определить какой ключ использовался при шифровании, если вам удалось отгадать хотя бы несколько букв можно, исходя из следующего свойства операции XOR:

X xor Key = Y
Y xor Key = X
X xor Y = Key

Например, если вы определили, что символу ‘A’ соответствует символ ‘p’, тогда ‘A’ xor ‘p’ = ‘1’ вернет ключ, которым текст шифровали.

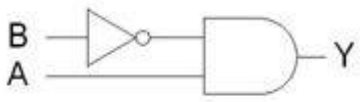
$$\begin{array}{rll} & 01110000 & \text{‘p’} \\ \oplus & \underline{01000001} & \text{‘A’} \\ & 00110001 & \text{‘1’} \end{array}$$

Для того чтобы спрятать X xor Y от посторонних глаз, или сократить размеры программы, помните, что

| Операция | Эквиваленты |
|---|---|
| X xor Y | (X or Y)–(X and Y) |
| | (X and (not Y))or((not X) and Y) |
| | (not(X or (not Y)))or(not((not X) or Y)) |
| | not((X or (not Y))and((not X) or Y)) |

### 9.1.5. Команда ANDN

(Логическое И-НЕ **(**штрих Шеффера**) = logical AND-NOT**)




| A | B | B' | Y |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| ANDNPD | | | | | | | ✓ | ✓ | ✓ | ✓ |
| ANDNPS | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ |
| PANDN | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Возможные варианты команды:*

```
MMX     pandn mmx,m64/mmx
SSE     andnps xmm,m128/xmm
SSE2    andnpd xmm,m128/xmm
        pandn xmm,m128/xmm
```

*Семантика команды:* операция штрих Шеффера над операндом *DEST размерностью байт, слово, двойное или учетверенное слово.*

*Алгоритм работы:*

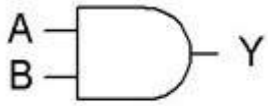
- Выполняется инверсия битов операнда *DEST* (регистр MMX/XMM), а затем побитовое «логическое И» над операндом *DEST* и операндом *SRC* (регистр MMX/XMM или переменная). Результат сохраняется в операнде *DEST*. Каждый бит операнда *DEST* устанавливается в 1, только если соответствующий бит операнда *SRC* равен 1, а операнда *DEST* – 0, иначе бит операнда *DEST* сбрасывается в 0.
- записать результат операции в операнд *DEST* (операнд *SRC* остается неизменным);
- установить флаги.

*Пример*:
```
mov mm0,00000A5A5h
mov mm7,000000FF0h
pandn mm0,mm7
```

| Переменная | Bin | Hex |
|---|---|---|
| mm0 | 00000000000000001010010110100101b | 00000A5A5h |
| ¬ mm0 | 11111111111111101011010010110101b | 0FFFF5A5Ah |
| mm7 | 00000000000000000000111111110000b | 000000FF0h |
| mm0=mm7∧(¬ mm0) | 00000000000000001010101001010101b | 000000A50h |

### 9.1.6. Команда TEST

(Логическое сравнение = **TEST operands**)




| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| | P | PII | K6 | 3D! | 3Mx+ | SSE | SSE2 | A64 | SSE3 | E64T |
|---|---|---|---|---|---|---|---|---|---|---|
| TEST | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Синтаксис команды:*

TEST <*SRC1*>,<*SRC2*>

*Возможные варианты команды:*

```
test   reg,reg/mem
test   mem,reg
test   reg/mem,imm
```

*Семантика команды:* операция логического сравнения операнда *SRC1* и *SRC2* размерностью байт, слово, двойное или учетверенное слово. Аналог операции логического умножения AND, но результат умножения никуда

не записывается (и поэтому команда TEST выполняется быстрее команды AND). Главное в команде – установка флагов.

*Алгоритм работы:*

- выполнить операцию логического умножения над операндом *SRC1* и *SRC2*: бит результата равен 1, если соответствующие биты операндов равны 1, в остальных случаях бит результата равен 0;
- установить флаги.

*Псевдокод*:

TEMP ←SRC1 AND SRC2

SF ←MSB(TEMP)
IF TEMP = 0
    THEN ZF ←1
ELSE ZF ←0
ENDIF;
PF←BitwiseXNOR(TEMP[0:7])
CF←0
OF←0; флаг AF имеет неопределенное значение

*Применение:* эту команду удобно использовать для получения информации о том, являются ли заданные биты операнда *SRC1* нулевыми. Для анализа результата используется флаг ZF, который равен 1, если результат логического умножения равен нулю:

*Пример*:
```
mov bh,1100b
test bh,0011b  ; bh=1100b ZF=1
test bh,1100b  ; bh=1100b ZF=0
```

Наиболее часто используется, чтобы узнать равен ли EAX нулю

```
test eax,eax/jz a1
```

| команда | эквивалент |
|---|---|
| test eax,ebx | push eax<br>and eax,ebx<br>pop eax |

## 9.2. Команды обработки бит

### 9.2.1. Команды сканирования бит

**Команда BSF**

(Побитное сканирование вперед = **B**it **S**can **F**orward)

*Синтаксис команды*:

BSF <*DEST*>,<*SRC*>

*Возможные варианты команды:*

```
bsf  reg,reg/mem
```

*Семантика команды*: для проверки наличия единичных битов в операнде *SRC*. Если первый единичный бит числа находится в N-ой позиции, значит само число кратно $2^{N-1}$.

*Псевдокод*:

IF SRC=0 THEN ZF←1; DEST имеет неопределенное значение.
ELSE
ZF←0
DEST←0
WHILE Bit(SRC,DEST)=0 DO
DEST←DEST+1
ENDDO
ENDIF

*Алгоритм работы:*

- просмотр битов операнда *SRC*, начиная с бита 0 и заканчивая битом 15/31, до тех пор, пока не встретится единичный бит;
- если встретился единичный бит, то флаг ZF устанавливается в 0 и в регистр операнда *DEST* записывается номер позиции, где встретился единичный бит. Диапазон значений зависит от разрядности операнда *SRC*: для 16-разрядного операнда – это 0...15; для 32-разрядного – это 0...31;
- если единичных битов нет, то флаг ZF устанавливается в 1.

*Применение: к*оманду BSF используют при работе на битовом уровне для определения позиции в операнде *SRC* крайних справа единичных битов.

Найдем номер крайнего справа бита в регистре EBX, содержащего единицу

```
MOV EBX,8004h
BSF ECX,EBX;ECX=2 в ECX номер крайнего правого единичного бита
```

что эквивалентно

```
MOV EBX,8004h
PUSH EBX
OR ECX,-1 ;ECX=-1
a1: SHR EAX,1;уменьшаем значение в EAX пока не встретим 1
INC ECX
JNC a1;ECX=2
POP EBX
```

Найдем номер крайнего справа бита в регистре EBX, содержащего ноль.

```
MOV EBX,8004h
NOT EBX
```

```
BSF ECX,EBX;ECX=0 в ECX номер крайнего правого единичного бита
NOT EBX;теперь в ECX номер крайнего правого нулевого бита
```

Выделение крайнего справа единичного бита в регистре EBX.

```
MOV EBX,8004h;1000.0000.0000.0100b
MOV EAX,EBX
NEG EAX;EAX=0FFFF7FFCh
AND EAX,EBX;EAX=4=0000.0000.0000.0100b
```

## Команда BSR

(Побитное сканирование назад = **Bit Scan Reverse**)

*Синтаксис команды:*

BSR <*DEST*>,<*SRC*>

*Возможные варианты команды:*

```
bsr  reg,reg/mem
```

*Семантика команды*: проверка наличия единичных битов в операнде SRC.

*Алгоритм работы:*

- просмотр битов *SRC*, начиная со старшего бита 15/31 и заканчивая битом 0 до тех пор, пока не встретится единичный бит;
- если встретился единичный бит, флаг ZF устанавливается в 0 и в регистр операнда *DEST* записывается номер позиции (отсчет осуществляется относительно нулевой позиции), где встретился самый старший единичный бит. Диапазон значений зависит от разрядности *SRC*: для 16-разрядного операнда это 0...15; для 32-разрядного – 0...31;
- если единичных битов нет, флаг ZF устанавливается в 1.

*Псевдокод*:

IF SRC=0 THEN ZF←1; DEST имеет неопределенное значение.
ELSE
    ZF←0
    DEST←OperandSize-1
    WHILE Bit(SRC,DEST)=0 DO
        DEST←DEST-1
    ENDDO
ENDIF

*Применение*: команду BSR используют при работе на битовом уровне для определения позиции крайних слева единичных битов.

Найдем номер крайнего слева бита в регистре EBX содержащего единицу.

```
MOV EBX,8004h
BSR ECX,EBX ;ECX=0Fh
```

что эквивалентно

```
     MOV EBX,8004h
     PUSH EBX
     MOV ECX,32
a1:  SHL EBX,1
     DEC ECX
     JNC a1    ;ECX=0Fh
     POP EBX
```

### 9.2.2. Команды проверки и модификации бит

#### Команда BT

(Проверка битов = **Bit Test**)

*Синтаксис команды:*

BT <*SRC*>,<*Index*>

*Возможные варианты команды:*

```
bt reg/mem,reg
bt reg/mem,imm
```

*Семантика команды*: извлечение значения заданного бита в флаг CF.

*Алгоритм работы:*

7. получить бит в операнде *SRC* по указанному номеру позиции в *Index*;
8. установить флаг CF согласно значению этого бита.

*Псевдокод*: CF←Bit(SRC, Index)

*Применение*: команду BT используют для определения значения конкретного бита в операнде *SRC*. Номер проверяемого бита задается содержимым операнда *Index* (значение числом из диапазона 0...31). После выполнения команды, флаг CF устанавливается в соответствии со значением проверяемого бита.

```
BT EBX,8 ; CF равен значению 8-го бита регистра EBX
JC M1 ;перейти на M1, если проверяемый бит равен 1
```

что эквивалентно

```
PUSH EBX
SHR EBX,9 ;CF равен значению 8-го бита регистра EBX
POP EBX
JC M1 ;перейти на M1, если проверяемый бит равен 1
```

#### Команда BTC

(Проверка бита и его инверсия = **Bit Test and Complement**)

*Синтаксис команды:*

BTC <*SRC*>,<*Index*>

*Возможные варианты команды:*

```
btc  reg/mem,reg
btc  reg/mem,imm
```

*Семантика команды*: извлечение значения заданного бита в флаг CF и изменение его значения в операнде *SRC* на обратное.
*Псевдокод*:

CF←Bit(SRC,Index)

Bit(SRC, Index)←NOT Bit(SRC, Index)

*Алгоритм работы*:

- получить значение бита с номером позиции *Index* в операнде *SRC*;
- инвертировать значение выбранного бита в операнде *SRC*;
- установить флаг CF исходным значением бита.

*Применение: к*оманда BTC используется для определения и инвертирования значения конкретного бита в операнде *SRC*. Номер проверяемого бита задается содержимым операнда *Index* (значение из диапазона 0...31). После выполнения команды флаг CF устанавливается в соответствии с исходным значением бита, то есть тем, которое было до выполнения команды.
Инвертирование 8-го бита регистра EBX:

```
MOV EBX,010011000b
BTC EBX,8      ;CF=0 и EBX=110011000b
```

что эквивалентно:

```
MOV EBX,110011000b
CLC;обнуляем CF
PUSH EBX
SHR EBX,9;CF равен измененному значению 8-го бита EBX
CMC;инвертируем CF, то есть устанавливаем его равным
POP EBX;исходному значению 8-го бита
```

## Команда BTR

(Проверка бита с его сбросом в 0 = **B**it **T**est and **R**eset)

*Синтаксис команды:*

BTR <*SRC* >,<*Index* >

*Возможные варианты команды:*

```
btr      reg/mem,reg
btr      reg/mem,imm
```

*Семантика команды*: извлечение значения заданного бита в флаг CF и изменение его значения на нулевое.

*Псевдокод*:

CF←Bit(SRC, Index)

Bit(SRC, Index)←0

*Алгоритм работы:*

- получить значение бита с указанным номером позиции в операнде *SRC*;
- установить флаг CF значением выбранного бита;
- установить значение исходного бита в операнде *SRC* в 0.

*Применение: к*оманда BTR используется для определения значения конкретного бита в операнде *SRC* и его сброса в 0. Номер проверяемого бита задается содержимым операнда *Index* (значение из диапазона 0...31). В результате выполнения команды флаг CF устанавливается в соответствии со значением исходного бита, то есть тем, что было до выполнения операции. Проверка состояния бита 8 регистра EBX и его сброс

```
MOV EBX,01001100h
BTR EBX,8 ;CF=1 и EBX=01001000h
```

что эквивалентно:

```
PUSH EAX
MOV EAX,EBX
AND EBX,0FFFFFEFFh;EBX=01001000h
SHR EAX,9;CF равен исходному значению 8-го бита EBX
POP EAX
```

## Команда BTS

(Проверка бита с его установкой в 1 = **Bit Test and Set**)

*Синтаксис команды:*

BTS <*SRC*>,<*Index*>

*Возможные варианты команды:*

```
bts   reg/mem, reg
bts   reg/mem, imm
```

*Семантика команды*: извлечение значения заданного бита операнда *SRC* в флаг CF и установка этого бита в единицу.

*Псевдокод*:

CF←Bit(SRC, Index)

Bit(SRC, Index)←1

*Алгоритм работы:*

- получить значение бита с указанным номером позиции в операнде *SRC*;
- установить флаг CF значением выбранного бита;
- установить значение исходного бита в операнде *SRC* в 1.

*Применение:* *к*оманда BTS используется для определения значения конкретного бита в операнде *SRC* и установки проверяемого бита в 1. Номер проверяемого бита задается содержимым операнда *Index* (значение из диапазона 0...31). После выполнения команды флаг CF устанавливается в соответствии со значением исходного бита, то есть тем, что было до выполнения операции.

Проверка состояния 0-го бита регистра EBX и его установка в 1

```
MOV EBX,01001000h
BTS EBX,0       ;CF=0 EBX=01001001h
```

что эквивалентно

```
PUSH EAX
MOV EAX,EBX
OR EBX,1;EBX=01001001h
SHR EAX,1;CF равен исходному значению 0-го бита EBX
POP EAX
```

## Работа с битами

Для обнуления крайнего справа единичного бита (например, 0101**1**000→0101**0**000) используется формула

```
X AND(X - 1)
```

Установка крайнего справа нулевого бита (01011**0**0→01011**1**0)

```
X OR(X+1)
```

Для выделения крайнего справа единичного бита (0101**1**000→000**1**000)

```
X AND(-X)
```

Для выделения крайнего справа нулевого бита (1010**0**111→0000**1**000)

```
(NOT X)AND(X+1)
```

Для выделения завершающих нулевых битов (например, 01011000→00000**111**) можно использовать одну из трех формул:

1) `(NOT X)AND(X-1)` или
2) `NOT(X OR(-X))` или
3) `(X AND(-X))-1`

Для выделения крайнего правого единичного бита и всех завершающих нулей (0101**1000**→0000**1111**)

```
X XOR(X-1)
```

Распространение вправо крайнего правого единичного бита (например 0101**1000**→0101**1111**)

```
X OR(X-1)
```

Обнуление крайней справа непрерывной подстроки единичных битов (010**111**00→010**000**00)

```
((X OR(X-1))+1)AND X
```

## Команды управления флагами микропроцессора

### *Установка флага CF*

*Синтаксис команды:*

STC (**SET Carry flag**)

*Семантика команды*: установка флага переноса CF в единицу.

*Псевдокод*:

EFLAGS.CF[bit 0]←1

### *Сброс флага CF*

*Синтаксис команды:*

CLC (**CLEAR Carry flag**)

*Семантика команды*: установка флага переноса CF в ноль.

*Псевдокод*:

EFLAGS.CF[bit 0]←0

### *Инвертировать флаг CF*

*Синтаксис команды:*

CMC (**COMPLEMENT Carry flag**)

*Семантика команды*: инвертирование флага переноса CF

*Псевдокод*:

EFLAGS.CF[bit 0]← NOT EFLAGS.CF[bit 0]

*Алгоритм работы:* Если CF=1, то после команды CMC флага переноса CF сбросится в ноль, если CF=0, то после команды CMC флага переноса CF установится в 1.

*Применение:* команды STC, CLC, CMC применяют, как правило, для установления признака успешного или не успешного выполнения подпрограмм.

### *Установка флага DF*

*Синтаксис команды:*

STD (**SET Direct flag**)

*Семантика команды*: установка флага направления DF в единицу.

*Псевдокод*:

EFLAGS.DF[bit 10]←1

### *Сброс флага DF*

*Синтаксис команды:*

CLD (**CLEAR Interrupt flag**)

*Семантика команды*: сброс флага направления DF в ноль.

*Псевдокод*:

EFLAGS.DF[bit 10]←0

*Применение:* в процессе циклического выполнения команд, значения в регистрах ESI и EDI автоматически модифицируются (уменьшаются или увеличиваются) в зависимости от длины элемента строки и значения флага направления DF. Если DF=0, значения в регистрах ESI и EDI увеличиваются (строка символов обрабатывается со стороны меньших адресов в сторону больших адресов). Если DF=1, значения в регистрах ESI и EDI уменьшаются (строка символов обрабатывается со стороны больших адресов в сторону меньших адресов).

### *Установка флага IF*

*Синтаксис команды:*

STI (**SET Interrupt flag**)

*Семантика команды*: установка флага разрешения прерывания IF в единицу, разрешая процессору распознавать *маскированные прерывания*. *Немаскированные прерывания* распознаются процессором всегда, независимо от значения флага прерывания IF.

*Псевдокод*:

EFLAGS.IF[bit 9]←1

### *Сброс флага IF*

*Синтаксис команды:*

CLI (**CLEAR Direct flag**)

*Семантика команды*: установка флага разрешения прерывания IF в ноль.

*Псевдокод*:

EFLAGS.IF[bit 9]←0

*Применение:* организация обработки критических участков программы, в которых прервать работу микропроцессора невозможно. Однако этим не стоит злоупотреблять, так как аппаратные прерывания использует и операционная система.

### *Установка флага ZF*

*Синтаксис команды:*

XOR AX,AX, или SUB AX, AX,

либо любая другая команда, которая даст нулевой результат.

*Семантика команды*: установка флага ZF в ноль.

*Псевдокод*:

EFLAGS.ZF[bit 6]←1

### *Установка флага TF*

*Синтаксис команды:*

INT 3

*Семантика команды*: установка флага TF в единицу.

*Псевдокод*:

EFLAGS.TF[bit 8]←1

*Применение:* пошаговое выполнение программы с целью ее выявления ошибок.

### *Одновременное изменение значения нескольких флагов*

Если необходимо изменить значения других флагов или одновременно изменить значения нескольких флагов используют следующие схемы:

а) для первых 8 флагов (SF, ZF, AF, PF, CF);

```
LAHF;загрузить значения младшего байта
;регистра EFLAGS в регистр AH
AND AH, маска ;сбросить необходимые флаги
;или OR AH, маска;или установить необходимые флаги
SAHF;загрузить младший байт регистра флагов
;значениями установленными в регистре AH
```

б) для первых 16 флагов (вышеперечисленные и NT, IOPL, OF, DF, IF, TF,);

```
PUSHF;передать содержимое EFLAGS на вершину стека
POP AX;прочесть слово, находящееся на вершине стека
;в один из регистров или в ячейку памяти
AND AX, маска;сбросить нужные флаги
;или OR AX, маска;  или установить нужные флаги
PUSH AX;  вновь передать содержимое регистра
;или ячейки памяти на вершину стека
POPF;прочитать слово с вершины стека в регистр  EFLAGS
```

в) для всех 32 флагов (все вышеперечисленные и AC, VM, RF).

```
PUSHFD
POP EAX
AND EAX, маска
;или OR EAX, маска
PUSH EAX
POPFD
```

### *Обобщаем. Как изменить (установить/сбросить) определенный бит в байте?*

Нумеровать биты принято с нуля, причем младший бит (нулевой) имеет маску 1, первый – 2, второй – 4, третий – 8 и т.д, т.е. маска = $\mathbf{2^n}$ (или **1 shl n**), где **n** – номер бита. Управлять битами можно с помощью инструкций **and**, **or**, **xor**, **btr**, **bts** и **btc**:

```
Mask = 1 shl n ;Маска
     and al,not Mask     ; Сбросить бит(ы)
     or al,Mask      ; Установить бит(ы)
     xor al,Mask     ; Инвертировать (изменить) бит(ы)
     and al,not (1 shl 2 + 1 shl 5);Сбросить второй и пятый биты
     btr al,n  ; Сбросить бит
     bts al,n  ; Установить бит
     btc al,n  ; Инвертировать (изменить) бит
```

Эти операции можно применять к операндам любого размера (байт, слово, двойное слово), находящимся как в регистре, так и в памяти.

Определить состояние бита можно с помощью инструкций **test** и **shr** (**ror**, **rcr**, **sar**) и инструкции **bt** (с помощью инструкций **btr**, **bts** и **btc** тоже можно определить состояние изменяемого бита):

```
Mask = 1 shl n ; Маска
     test al,Mask   ; Проверить состояние бита
     jz Reset  ; переход, если бит сброшен
     jnz Set   ; переход, если бит установлен

     test al,(1 shl 2+1 shl 5);Проверить состояние 2-ого и 5-ого битов
     jz Reset  ; переход, если ОБА бита сброшены
     jnz Set   ;переход, если ХОТЯ БЫ ОДИН из битов установлен

     shr al,n+1     ;Проверить бит n (регистр AL изменяется!)
     jnc Reset ;переход, если бит сброшен
     jc   Set; переход, если бит установлен

     bt al,n   ; Проверить бит n
     jnc Reset ; переход, если бит сброшен
     jc Set    ; переход, если бит установлен

     bts al,n  ; Проверить бит n и установить его
     jnc Reset ; переход, если бит был сброшен
     jc Set    ; переход, если бит был установлен
```

## Контрольные вопросы и упражнения

1. Предположим, что регистр BL содержит 11100011b и переменная по имени BOOLDOG содержит 01111001b. Напишите программы, определяющие воздействие на регистр BL следующих команд:

   а) XOR BL,BOOLDOG; б) AND BL,BOOLDOG;
   в) OR BL,BOOLDOG; г) NOT BL;
   д) XOR BL,BL; е) XOR BL,11111111b;
   ж) OR BL,11111111b; з) AND BL,00000000b.

2. Предположим, что регистр BL содержит 11100011b. Укажите возможные способы, которыми можно инвертировать содержимое регистра BL.
3. Составить программу, которая по состоянию 0-го и 1-го битов регистра EDX:

   а) сбрасывала 0-й и 1-й биты, если они оба установлены;
   б) устанавливала 0-й и 1-й биты, если они оба сброшены;
   в) не изменяла 0-й и 1-й биты в остальных случаях.

4. Составить программу, которая после проверки 2 и 4 битов переменной в ячейке памяти YORK передавала бы управление на метку ERR12, если оба бита установлены, передавала управление на метку ERR1, если установлен бит 2 – передавала управление на метку ERR2, если установлен бит 4 и продолжалась бы, если оба бита сброшены.

## СОДЕРЖАНИЕ

Учебное пособие

**М.Ю. Смоленцев**

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА ДЛЯ 32/64-РАЗРЯДНЫХ МИКРОПРОЦЕССОРОВ СЕМЕЙСТВА 80X86

## Часть 1

Редактор *Н.Е. Кильдишева*
Подписано в печать 30.11.2009.
План 2009 г.
Усл. печ. л. 12. Уч.-изд. л. 12,95.
Тираж 100 экз. Заказ

Типография ИрГУПСа, г. Иркутск,
ул. Чернышевского, 15