ИВАН СКЛЯРОВ

ИЗУЧАЕМ ASSEMBLER ЗА 7 ДНЕЙ

WWW.SKLYAROFF.RU

ИВАН СКЛЯРОВ

Изучаем Assembler за 7 дней

2010

Содержание

Введение

Существует множество мифов, связанных с ассемблером, главным из которых является то, что этот язык очень сложен в изучении. На самом деле, ассемблер очень простой язык программирования, а появление мифа о сложности его изучения связно с отсутствием нормальных учебников по этому языку. Это я вам говорю по собственному опыту изучения ассемблера. Не хочу умалять писательских талантов других авторов, но большинство из тех увесистых книг по ассемблеру, которые вы можете увидеть на полках книжных магазинов, к сожалению, не подходят для начинающих. Мне самому в свое время пришлось с большим трудом осилить несколько таких "кирпичей" прежде чем я почувствовал, что начал более-менее разбираться в ассемблере. Поэтому я поставил себе цель написать такую книгу, после прочтения которой, программы на ассемблере смог бы писать даже человек далекий от программирования. В этой книге я учел и разобрал все те сложности, которые возникали у меня у самого на начальном этапе. Кроме того, я расположил материал в такой последовательности, в которой, по моему мнению, его лучше всего изучать.

В то же время эта книга не является всеобъемлющим справочником по ассемблеру, поэтому в списке литературы я привел названия тех книг, которые стоит прочитать для более глубокого изучения ассемблера. После прочтения моей книги, у вас не должно уже возникнуть проблем с пониманием любых сложных материалов по ассемблеру.

Второй популярный миф, связанный с ассемблером, состоит в том, что якобы ассемблер в наше время никому не нужен, и тратить время на его изучение не имеет смысла. Да, современные языки программирования высокого уровня, такие как Delphi, Visual C++, Visual C# и др. позволяют быстро и легко писать программы любой сложности. Долгое время обоснованием для выбора ассемблера являлось написание программ, требующих минимального размера и максимального быстродействия. Но в наше время гигабайтных модулей памяти и терабайтных жестких дисков и эта способность ассемблера не столь актуальна, к тому же компиляторы современных языков высокого уровня "научились" создавать достаточно быстрый и компактный код. Однако я считаю, изучать ассемблер необходимо каждому IT-специалисту (тем более программисту), потому что только ассемблер позволяет узнать устройство процессора и по-настоящему понять, как работает компьютер на уровне его "железа".

Ну и конечно ассемблер навсегда останется необходимым для хакеров, крэкеров и прочих любителей покопаться во внутренностях чужих программ.

Уверен, вы не зря потратите 7 дней на изучение этого самого древнего и замечательного языка программирования.

В одном архиве с книгой содержатся все исходные коды приведенных листингов.

ДЕНЬ 1

Знакомство с архитектурой компьютера

Если вы уже изучали какой-нибудь язык программирования высокого уровня (Basic, Pascal, Си, C#, ...), то, наверняка, помните, что его изучение начиналось с освоения основных команд и написания первой простейшей программы. С ассемблером так сразу не получится. Но это совсем не потому, что ассемблер такой сложный, а потому что программы на ассемблере напрямую манипулируют устройствами компьютера, в первую очередь процессором и памятью. Отсюда неслучайно ассемблер называют языком низкого уровня. Языки высокого уровня скрывают от программиста все манипуляции с компьютерным "железом".

Таким образом, чтобы научиться программировать на ассемблере, необходимо знать *архитектуру компьютера*. Поэтому материал этого дня очень важен и насколько хорошо вы его усвоите, зависит ваше будущее как программиста на ассемблере. Возможно что-то вам будет понятно не сразу, но не переживайте на следующих днях мы с вами все проясним.

1.1. Что такое архитектура компьютера

Четкого определения, что такое архитектура компьютера не существует, поэтому я приведу наиболее известное и симпатичное мне:

Архитектура компьютера – это логическая организация, структура и ресурсы компьютера, которые может использовать программист.

Архитектура компьютера включает в себя архитектуры отдельных устройств, входящих в компьютер. Хотя компьютер состоит из многих внешних и внутренних устройств, но реально программисту на ассемблере приходится работать только с тремя устройствами компьютерной системы: *процессором*, *памятью* и *портами ввода-вывода*. В сущности, эти три устройства определяют работу всего компьютера и работу всех внешних устройств подключенных к нему. Все эти три устройства соединены между собой при помощи трех основных шин: *шиной данных (ШД)*, *шиной адреса (ША)* и *шиной управления (ШУ)* (рис. 1.1). Шины это просто набор проводников по которым передаются цифровые сигналы (машинные коды) от процессора к памяти, от процессора к портам ввода-вывода, и обратно от этих устройств к процессору. Все три шины вместе образуют *системную шину* или ее еще называют *магистраль*.

Существует еще такое понятие как *микроархитектура*. Если архитектура это программно-видимые свойства устройства, то микроархитектура – внутренняя реализация архитектуры. Для одной и той же архитектуры разными производителями могут применяться существенно различные микроархитектурные реализации с различной производительностью и стоимостью.

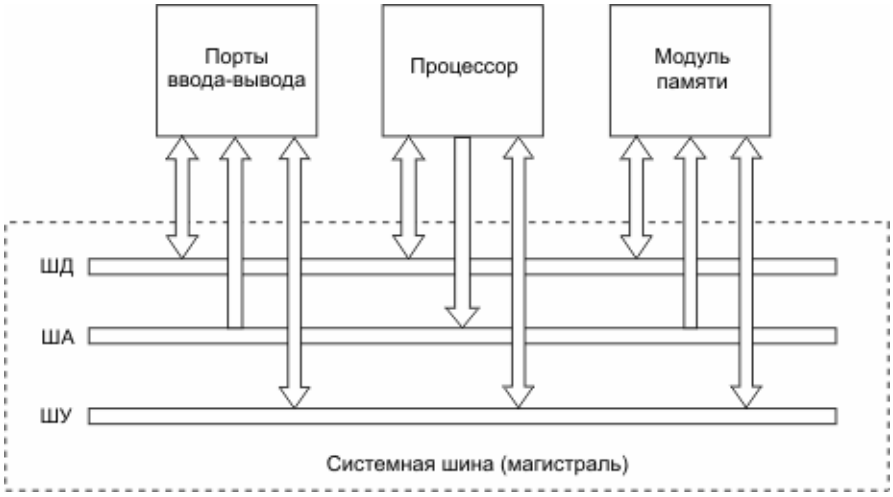

Рис. 1.1. Структурная схема компьютера

Например, к архитектуре процессора относятся регистры и набор инструкций (команд), которые может использовать программист в своей программе (далее в книге мы с ними познакомимся подробнее). А к микроархитектуре процессора – способы конвейеризации, распараллеливания, буферизации вычислений и другие технологии, предназначенные для увеличения скорости работы процессора. В микроархитектуре процессора тоже может использоваться свой набор команд, но он обычно недоступен программисту.

Мы в этой книге почти не будем касаться микроархитектуры, т. к. программист не имеет возможности работать с ней напрямую, к тому же, как говорилось, она отличается у различных производителей и часто существенно меняется при переходе от одного поколения процессоров к другому.

Но прежде чем перейти к более подробному изучению архитектуры процессора, памяти и портов ввода-вывода, необходимо остановиться на системах счисления, а также битах и байтах, потому что они тесно связаны с архитектурой компьютера.

1.2. Системы счисления

Слово "компьютер" (computer) с английского языка переводится как "вычислитель", т. е. машина для проведения вычислений. И это полностью соответствует действительности, т. к. на уровне "железа" компьютер выполняет только простейшие арифметические операции с числами, такие как сложение и умножение. В СССР, да и в современной России часто компьютер прямо так и называют: ЭВМ ("электронная вычислительная машина").

Сердцем компьютера является процессор, называемый часто *центральным процессором* (ЦП) или *микропроцессором*. Именно центральный процессор выполняет все вычисления.

Так исторически сложилось, что практически все цифровые микросхемы, в том числе компьютерные процессоры, работают только с двумя разрешенными уровнями напряжения. Один из этих уровней называется уровнем логической единицы (или единичным уровнем), а другой — уровнем логического нуля (или нулевым уровнем). Чаще всего логическому нулю соответствует низкий уровень напряжения (от 0 до 0,4 В), а логической единице — высокий уровень (от 2,4 до 5 В)[1]. Два уровня напряжения было выбрано исключительно из-за простоты реализации.

Таким образом, можно образно представлять, что в электронной цепи компьютера "бегают" только цепочки ноликов и единичек. За этими цепочками нулей и единичек закрепилось название *машинные коды*. Точно также можно представлять, что в память компьютера, а также на магнитные, оптические и прочие носители записываются нолики и единички, которые в совокупности составляют хранимую информацию.

[1] Диапазоны приведены для процессоров с напряжением питания 5 В (в современных процессорах оно как правило ниже).

То есть компьютер способен воспринимать только нолики и единички, а для нас (людей) эти нолики и единички представляются через устройства вывода (дисплеи, принтеры, звуковые колонки и пр.) в виде текста, графических изображений и звуков.

Следует отметить, что были попытки сделать ЭВМ на основе троичной логики (например, отечественная ЭВМ "Сетунь", 1959 г.) и даже на основе десятичной логики (американская ЭВМ "Марк-1", 1943 г.). Но распространения они не получили, т. к. их устройство сложнее машин на основе двоичной логики, а потому они дороже и менее надежны. К тому же все, что осуществимо на машинах с троичной и десятичной логикой (и вообще любой недвоичной логикой), то осуществимо и на машинах с двоичной логикой.

Так как компьютер способен воспринимать только два управляющих сигнала: 0 и 1, то и любая программа должна быть ему представлена только в двоичных кодах, т. е. в *машинных кодах*. В старые добрые времена операторы первых ЭВМ программировали напрямую в машинных кодах, переключая специально предусмотренные для этого тумблеры, или пробивали двоичные коды на перфолентах и перфокартах, которые затем считывала ЭВМ и выполняла операции согласно этим кодам. Перфокарта представляла собой картонный прямоугольный лист, на который была нанесена цифровая сетка (рис. 1.2). Информация на перфокарты наносилась оператором путем пробивки отверстий в нужных местах цифровой сетки на перфокарте с помощью специального электромеханического устройства – перфоратора. Наличие отверстия означало код 1, а его отсутствие – код 0. Информация считывалась с перфокарты в процессе перемещения ее в специальном устройстве считывателе. В считыватель обычно подавалась сразу стопка перфокарт с нанесенной информацией.

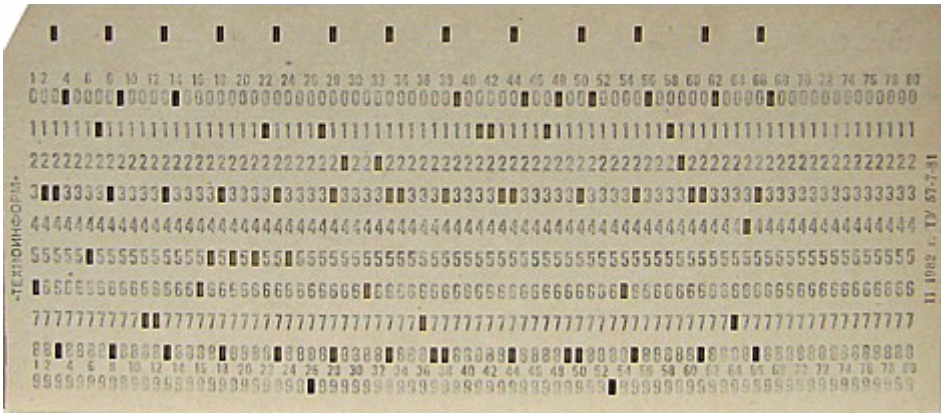

Рис. 1.2. Перфокарта с пробитыми отверстиями

Однако записывать и запоминать огромные двоичные цепочки, первым программистам было неудобно, поэтому они стали вместо двоичной системы использовать другие системы счисления, например десятичную, восьмеричную или шестнадцатеричную. Сравните: двоичное число 11001000 будет представлено в десятичном виде как 200, а в восьмеричной и шестнадцатеричной соответственно как 310 и C8.

Здесь еще раз отмечу: недвоичные системы счисления первые программисты стали использовать исключительно для личного удобства. Компьютер не способен воспринимать десятичные, шестнадцатеричные или восьмеричные числа, а *только и только* двоичные коды!

> ***Примечание***
> Надеюсь, вы знаете, что в десятичной системе счисления используется десять цифр (от 0 до 9), в восьмеричной системе счисления используется только восемь цифр (от 0 до 7). В шестнадцатеричной системе счисления используется 16 цифр, где в качестве первых десяти цифр применяются цифры десятичной системы, а недостающие шесть цифр заменяются буквами латинского алфавита, т. е. полный набор шестнадцатеричных цифр выглядит так: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Как в литературе, так и в программах на ассемблере, для обозначения системы счисления, в которой записано число, принято ставить в конце числа букву **b (Bin)** –

для двоичного, **o (Oct)** – для восьмеричного, **h (Hex)** – для шестнадцатеричного числа. Для обозначения десятичной системы счисления обычно не используется никакой буквенной приставки или в редких случаях ставится d (Dec). Одно и то же число, записанное в различных системах счисления, будет выглядеть так:

200**d** = 11001000**b** = 310**o** = C8**h**

Таким образом, операторы первых ЭВМ стали составлять свои программы в более удобной системе счисления (восьмеричной, шестнадцатеричной или другой), а потом переводить их в двоичный машинный код. Наибольшее распространение у первых программистов из всех систем счисления получила шестнадцатеричная система счисления, которая до сих пор является основной в компьютерном мире. Вас, наверное, интересует, почему получила распространение именно шестнадцатеричная система счисления? Все просто, в отличие от других систем счисления перевод из шестнадцатеричной системы счисления в двоичную систему и обратно осуществляется очень легко — вместо каждой шестнадцатеричной цифры, подставляется соответствующее четырехзначное двоичное число, например:

486h = 10010000110b, где 4h = 0100b, 8h = 1000b, 6h = 0110b

Для сравнения лишь покажу, как перевести десятичное число в двоичное (возьмем для примера число 200). Для этого надо его делить на 2 и записывать остаток справа налево:

200/2 = 100 (остаток 0)
100/2 = 50 (остаток 0)
50/2 = 25 (остаток 0)
25/2 = 12 (остаток 1)
12/2 = 6 (остаток 0)
6/2 = 3 (остаток 0)
3/2 = 1 (остаток 1)
1/2 =0 (остаток 1)
Результат: 11001000b

А для обратного перевода двоичного числа в десятичное, необходимо сложить двойки в степенях, соответствующих позициям, где в двоичном стоят единицы. Пример:

$11001000b=1\times2^7+1\times2^6+0\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+0\times2^0=128+64+8=200$

Во времена древних ЭВМ еще не существовало калькуляторов и программистам приходилось все численные преобразования делать в уме, либо с помощью карандаша и листка бумаги, поэтому использование шестнадцатеричной системы значительно облегчило этот труд.

Мы не будем изучать правила преобразования чисел из одной системы счисления в другую, т. к. надеюсь, вам не придется это делать вручную. Лучше доверить эту задачу любому инженерному калькулятору, например стандартному Калькулятору в операционной системе Windows (вызывается **Пуск -> Программы -> Стандартные -> Калькулятор**), для этого в его меню **Вид** должен быть выставлен режим "Инженерный".

Если все-таки вы желаете научиться делать перевод чисел вручную, то советую почитать любой учебник по информатике.

Думаю понятно, что хотя шестнадцатеричная система облегчила работу с машинными кодами, но создавать программу в шестнадцатеричном виде все равно очень не просто. В итоге родился язык ассемблера, который давал возможность

писать программы на более понятном человеку языке и в то же время позволял легко переводить их в машинный код.

Язык ассемблера прозвали низкоуровневым языком, потому что он максимально приближен к машинному языку, а значит к "железу" компьютера. После языка ассемблера стали появляться высокоуровневые языки, такие как Бейсик, Паскаль, Фортран, Си, C++ и пр. Они еще более понятны человеку, но преобразование в машинный код высокоуровневых программ значительно сложнее, из-за чего размер кода, как правило, получается большим и менее быстрым по сравнению с ассемблерными программами.

Например, команда на Си: **x = x + 33;** (прибавить к переменной **x** число **33** и результат сохранить в **x**) на языке ассемблера будет выглядеть так: **add ax,33** (слово **add** с англ. переводится как "добавить"). В свою очередь в шестнадцатеричном виде эта команда будет представлена следующим образом: **83h C0h 21h**, а в машинном (двоичном) коде, так: **100000111100000000100001**.

Если операторы первых ЭВМ переводили свои программы в машинный код вручную, то сейчас эту работу выполняют специальные программы — трансляторы (англ. translator — переводчик). Для языков высокого уровня транслятор принято называть компилятором (англ. compiler — составитель, собиратель). Для языка ассемблера обычно тоже не используется слово транслятор, а говорят просто: "ассемблер". Таким образом, ассемблером называют, как язык программирования, так и транслятор этого языка.

Соответственно процесс работы ассемблера называют *ассемблированием*. Процесс работы компилятора называют *компилированием*. Процесс обратный ассемблированию, т. е. преобразование машинного кода в программу на языке ассемблера называют *дизассемблированием*.

Рассмотрим еще очень важные понятия, непосредственно касающиеся архитектуры компьютера: *биты* и *байты*.

1.3. Биты и байты

Цифра в двоичной арифметике называется *разрядом* (или точнее "двоичным разрядом") и может принимать значение ноль или единица. В компьютерном мире вместо разряда часто употребляют название *бит*. Далее в этой книге я буду писать либо бит, либо разряд — вы должны понимать, что это одно и то же.

Таким образом, минимальной единицей информации в компьютерной системе является бит, который может принимать только значение 0 или 1. Однако минимальным объемом данных, которым позволено оперировать любой компьютерной программе является не бит, а *байт*. Байт состоит из восьми бит. Если программе нужно изменить значение только одного бита, то она все равно должна считать целый байт, содержащий этот бит. Биты в байте нумеруются справа налево от 0 до 7, при этом нулевой бит принято называть *младшим*, а седьмой — *старшим* (рис. 1.3).

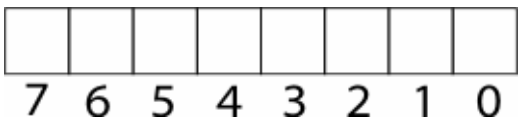

Рис. 1.3. Байт

Так как в байте всего восемь бит, а бит может принимать только два значения, то простой арифметический подсчет показывает, что байт может принимать до 2^8=256 различных значений. Поэтому в байте могут быть представлены целые числа в диапазоне от 0 до 255, или числа со знаком от –128 до +127. Подробнее о представлении чисел на ассемблере мы еще подробно поговорим на пятом дне нашего курса.

Однако не только байтами может оперировать компьютерная программа, но и более крупными единицами данных — *словами*, *двойными словами* и *учетверенными словами*. Слово состоит из двух байт, при этом биты с 0 по 7 составляют *младший*

байт в слове, а биты с 8 по 15 — *старший* (рис. 1.4). Понятно, что слово может принимать до 2^{16}=65536 различных значений.

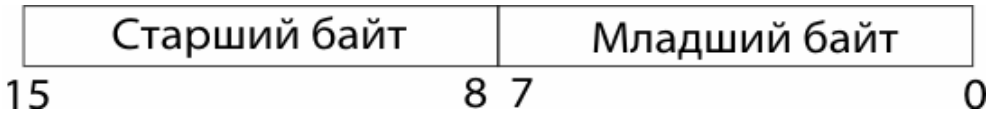

Рис. 1.4. Слово

Двойное слово, как следует из самого названия, состоит из двух слов или четырех байт, а значит из 32-х бит, а два двойных слова составляют учетверенное слово (64 бита).

Существует еще более крупная единица, которая называется *параграф* и представляет собой 16 смежных байт.

1.4. Фон-неймановская архитектура

Подавляющее большинство современных вычислительных машин, в том числе IBM PC-совместимые компьютеры, представляют собой реализацию так называемой фон-неймановской архитектуры. Эту архитектуру предложил американский ученый венгерского происхождения Джордж фон Нейман в 1945 году[2].

Машина фон Неймана состояла из следующих устройств:

- АЛУ – арифметико-логическое устройство для выполнения арифметических и логических операций;
- ОП – оперативная память для хранения выполняющейся в данный момент программы;
- УВВ – устройства ввода-вывода для ввода и вывода информации;
- УУ – управляющее устройство, которое организует работу компьютера следующим образом:
  - помещает в оперативную память коды программы из устройств ввода;
  - считывает из ячейки оперативной памяти и организует выполнение первой команды программы;
  - определяет очередную команду и организует ее выполнение;
  - постоянно синхронизирует работу устройств, имеющих различную скорость выполнения операций, путем приостановки выполнения программы.

В современных компьютерах роль АЛУ и УУ выполняет центральный процессор.

Архитектура фон-неймановской машины основана на следующих фундаментальных принципах (на этих же принципах работают все современные компьютеры, поэтому вам их нужно хорошо знать):

- **Принцип программного управления.** Машиной управляет заранее подготовленная программа, представляющая собой последовательность инструкций (команд), расположенных в основной памяти машины линейно друг за другом. Команды исполняются последовательно, в порядке их записи. Такая последовательность выполнения команд называется *естественной*. Естественный порядок выполнения команд может нарушаться командами перехода (рис. 1.5).
- **Принцип хранимой программы.** Для выполнения программы она должна быть предварительно помещена в оперативную память машины, а затем инициировано выполнение первой команды. Команды выбираются центральным процессором из оперативной памяти автоматически и интерпретируются в соответствии с

[2] Существует версия, что авторство идеи принадлежит не ему, а разработчикам сверхсекретного в то время компьютера ENIAC Джону Маучли (J. Mauchly) и Джону Эккерту (J. Eckert), у которых Нейман проходил стажировку. Учитывая это, в настоящее время данную архитектуру все чаще называют принстонской, по названию университета, в котором работали Маучли и Эккерт.

принципом программного управления. Команды и данные программы хранятся одинаково в единой области памяти. Что считать командами, а что данными процессор определяет неявно в зависимости от использования информации.

Существуют и другие архитектуры отличные от фон-неймановской, например гарвардская архитектура. В гарвардской архитектуре команды и данные размещаются в разных видах памяти: в памяти команд и в памяти данных, соответственно. Это обеспечивает более высокую надежность работы машины, т. к. исключается возможность обращения с командами как с данными и наоборот. В фон-неймановской архитектуре из-за того, что данные и команды располагаются совместно в единой памяти, возможны ситуации, когда процессор начинает интерпретировать данные как команды или команды как данные, что обычно приводит к сбою. С другой стороны совместное размещение команд и данных позволяет писать более гибкие программы.

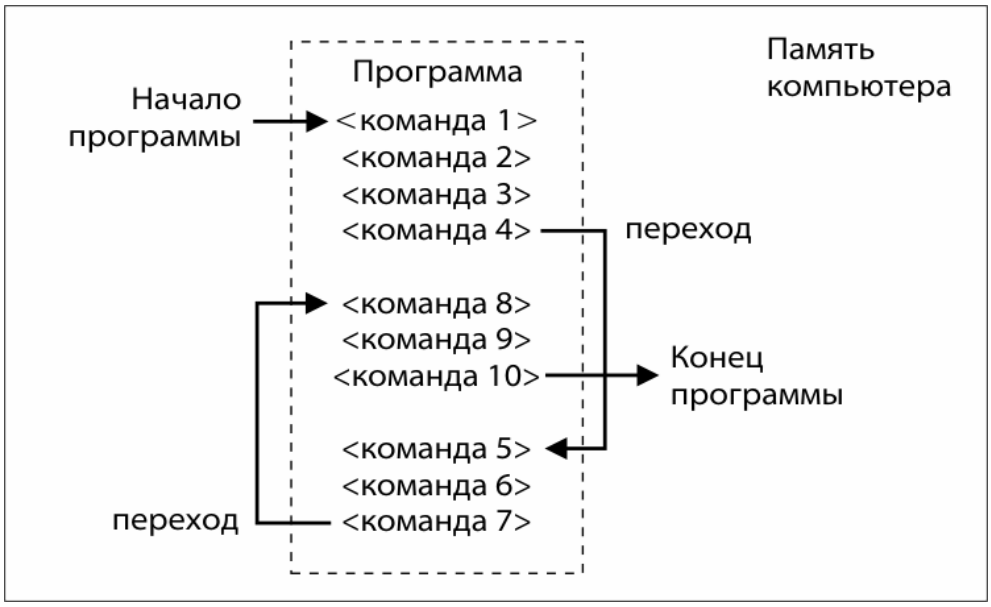

Рис. 1.5. Порядок выполнения команд программы

Так или иначе, как уже было сказано, работа подавляющего большинства современных компьютеров основана именно на фон-неймановских принципах, включая и сложные многопроцессорные комплексы. На рис. 1.1 показана архитектура компьютера соответствующая фон-неймановской архитектуре.

Таким образом, согласно фон-неймановской архитектуре программа транслированная (откомпилированная или ассемблированная) в машинный код перед непосредственным выполнением **всегда** помещается в память. Программа в памяти — это просто последовательность инструкций (команд) расположенных линейно друг за другом (в двоичном виде, разумеется). Процессор считывает команды из памяти в порядке их записи и выполняет. В зависимости от считанной команды процессор выполняет определенные действия, это могут быть какие-либо вычисления или сигнал на какой-либо порт ввода-вывода, например указание жесткому диску переместить головку в определенное положение.

Как мы уже говорили, работа компьютера определяется тремя основными устройствами: *процессором*, *памятью* и *портами ввода-вывода*. Чтобы начать программировать на ассемблере, нужно хорошо разобраться с архитектурой каждой из этих трех частей компьютера. Рассмотрим все эти три части по порядку.

1.5. Процессор

Мы будем рассматривать только архитектуру процессоров IA-32 (Intel Architecture 32 bit) – 32-разрядные процессоры семейства x86, т. к. это в настоящее время основная архитектура процессоров используемых на подавляющем большинстве компьютеров.

Почему эти процессоры называются 32-разрядными, и что означает сокращение "x86" будет рассказано ниже.

Процессоры делятся на поколения. История семейства x86 фирмы Intel началась с 16-разрядного процессора 8086, который относится к первому поколению (отсюда и сокращение x86). Начиная с процессора 80386 (третье поколение) все последующие модели процессоров являются 32-разрядными. 64-х разрядные Intel-совместимые процессоры (IA-64) пока еще не получили массового распространения. В приложении 1 вы можете увидеть деление процессоров Intel на поколения, а также года их выпусков, основные характеристики и общепринятые обозначения.

> ***Примечание***
> Все детали архитектуры процессоров IA-32 вы можете найти в фирменной документации от Intel. Ее можно скачать бесплатно на английском языке в формате PDF с Web-сервера поддержки разработчиков фирмы Intel по адресу: **http://developer.intel.com/products/processor/manuals/index.htm**. Начать изучение можно с руководства *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*.

1.5.1. Режимы работы процессора

Процессор архитектуры IA-32 может работать в одном из пяти режимов и переключаться между ними очень быстро:

1. **Реальный (незащищенный) режим (real address mode)** — режим, в котором работал процессор 8086. В современных процессорах этот режим поддерживается в основном для совместимости с древним программным обеспечением (DOS-программами).
2. **Защищенный режим (protected mode)** — режим, который впервые был реализован в 80286 процессоре. Все современные операционные системы (Windows, Linux и пр.) работают в защищенном режиме. Программы реального режима не могут функционировать в защищенном режиме.
3. **Режим виртуального процессора 8086 (virtual-8086 mode, V86)** — в этот режим можно перейти только из защищенного режима. Служит для обеспечения функционирования программ реального режима, причем дает возможность одновременной работы нескольких таких программ, что в реальном режиме невозможно. Режим V86 предоставляет аппаратные средства для формирования виртуальной машины, эмулирующей процессор 8086. Виртуальная машина формируется программными средствами операционной системы. В Windows такая виртуальная машина называется VDM (Virtual DOS Machine — виртуальная машина DOS). VDM перехватывает и обрабатывает системные вызовы от работающих DOS-приложений.
4. **Нереальный режим (unreal mode, он же big real mode)** — аналогичен реальному режиму, только позволяет получать доступ ко всей физической памяти, что невозможно в реальном режиме.
5. **Режим системного управления System Management Mode (SMM)** используется в служебных и отладочных целях.

При загрузке компьютера процессор всегда находится в реальном режиме, в этом режиме работали первые операционные системы, например MS-DOS, однако современные операционные системы, такие как Windows и Linux переводят процессор в защищенный режим. Вам, наверное, интересно, что защищает процессор в защищенном режиме? В защищенном режиме процессор защищает выполняемые программы в памяти от взаимного влияния (умышленно или по ошибке) друг на друга, что легко может произойти в реальном режиме. Поэтому защищенный режим и назвали защищенным.

1.5.2. Регистры процессора

В процессоре содержатся быстродействующие ячейки памяти, называемые *регистрами*, которые может и должна использовать любая программа. Каждый регистр имеет свое уникальное имя. Именно с помощью регистров программисты манипулируют процессором в своих программах на ассемблере. Начиная с 386 процессора, регистры делятся на следующие группы:

- 16 пользовательских регистров;
- 16 системных регистров;
- 13 регистров для работы с мультимедийными приложениями (MMX) и числами с плавающей запятой (FPU/NPX);
- В современных процессорах (PIII, P4) имеются дополнительные регистры: XMM (расширение SSE/SSE2).

В последующих поколениях процессоров следует ожидать только увеличения числа регистров, как это происходило до сих пор. Как уже говорилось, в процессоре имеются невидимые для программиста регистры, входящие в микроархитектуру процессора, которые процессор использует только для собственных нужд.

Пользовательские регистры это основные регистры, которые использует программист на ассемблере. Системные регистры используются в защищенном режиме ассемблера. Регистры FPU, MMX и XMM необходимы для ускорения вычислений и чаще всего используются в графических приложениях (в компьютерных играх).

1.5.2.1. Пользовательские регистры

Пользовательские регистры разделяются на *регистры общего назначения*, *сегментные регистры*, *регистры флагов и указателя команд* (рис. 1.6)

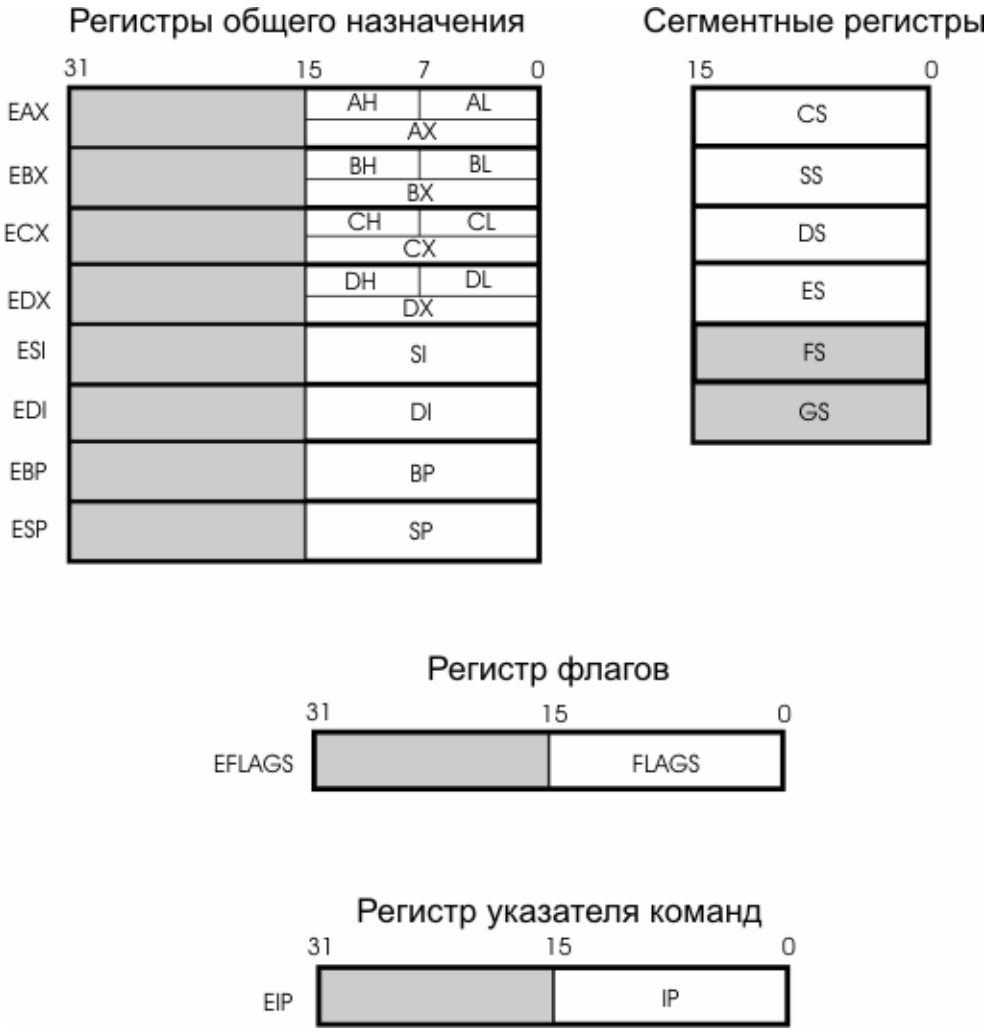

Рис. 1.6. Пользовательские регистры

1.5.2.1.1. Регистры общего назначения

Наиболее интенсивно используемыми в процессоре являются регистры общего назначения. В процессорах первого поколения регистры общего назначения были 16-

разрядными. Начиная с третьего поколения (с процессора 80386) регистры общего назначения стали 32-разрядными. Расширения, которые появились в 32-разрядных процессорах, выделены на рис. 1.6 серым цветом.

Именно из-за того, что регистры общего назначения являются 32-разрядными современные Intel-совместимые процессоры и называются 32-разрядными. Аналогично процессоры первого поколения назывались 16-разрядными потому, что имели 16-разрядные регистры общего назначения. Соответственно 64-разрядные процессоры Intel имеют 64-разрядные регистры общего назначения.

Из-за совместимости с процессорами первых поколений регистры общего назначения можно использовать, как полностью 32 бита (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), так и только младшую половину 16 бит (AX, BX, CX , DX, SI, DI, BP, SP). В свою очередь младшая половина в некоторых регистрах общего назначения также может использоваться частями по 8 бит (AH, AL, BH, BL, CH, CL, DH, DL). Как видно названия 32-битных регистров отличаются от 16-битных только приставкой E (Extended — расширенный).

Большинство регистров общего назначения используются при программировании без ограничений для любых целей. Однако в некоторых случаях вводится жесткое ограничение. Далее идет краткое описание всех регистров общего назначения:

- EAX/AX/AH/AL (Accumulator register) — аккумулятор. В основном используется для хранения любых промежуточных данных. Только в некоторых командах использование этого регистра обязательно.
- EBX/BX/BH/BL (Base register) — база. В основном используется для хранения любых промежуточных данных. Некоторые команды используют этот регистр при так называемой адресации по базе.
- ECX/CX/CH/CL (Count register) — счетчик. В основном используется для хранения любых промежуточных данных. Использование этого регистра обязательно только в командах организации цикла (повторяющихся действий).
- EDX/DX/DH/DL (Data register) — регистр данных. В основном используется для хранения любых промежуточных данных. Только в некоторых командах использование этого регистра обязательно.
- ESI/SI (Source Index register) — индекс источника. Используется в цепочечных операциях (обычно цепочкой является строка символов) и содержит адрес элемента в цепочке-приемнике.
- EDI/DI (Destination Index register) — индекс приемника (получателя). Используется в основном в цепочечных операциях (цепочкой обычно является строка символов) и содержит текущий адрес в цепочке-приемнике.
- EBP/BP (Base Pointer register) — регистр указателя базы кадра стека. Предназначен для организации произвольного доступа к данным внутри стека.
- ESP/SP (Stack Pointer register) — регистр указателя стека. Содержит указатель вершины стека.

1.5.2.1.2. Сегментные регистры

В реальном режиме работы процессора процессор может аппаратно делить программу в памяти на 3 части, которые прозвали *сегментами*, а сегментные регистры соответственно предназначены для доступа к этим сегментам:

1. **Сегмент кода.** В этом сегменте содержатся машинные команды. Для доступа к этому сегменту служит регистр CS (code segment register) — сегментный регистр кода.
2. **Сегмент данных.** Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр DS (data segment register) — сегментный регистр данных.
3. **Сегмент стека.** В этом сегменте содержится стек. Для доступа к этому сегменту служит регистр SS (stack segment register) — сегментный регистр стека.

Если программисту недостаточно одного сегмента данных адресуемого регистром DS, то он может задействовать в своей программе дополнительные сегменты данных с помощью сегментных регистров ES, GS, FS (extension data segment registers).

1.5.2.1.3. Регистр флагов и указателя команд

- EIP/IP (Instruction Pointer register) — указатель команд. 32/16-разрядный регистр, который содержит адрес следующей машинной команды в памяти, подлежащей выполнению.
- EFLAGS/FLAGS (flag register) — регистр флагов. Каждый отдельный бит этого 32/16-разрядного регистра называется флагом. Говорят флаг установлен, если соответствующий разряд содержит 1, и сброшен, если разряд содержит 0.

На рис. 1.7 показано содержимое регистра EFLAGS, а в табл. 1.1 перечислены названия и назначение каждого флага.

Когда мы приступим непосредственно к программированию на ассемблере, назначение большинства флагов станет более понятно.

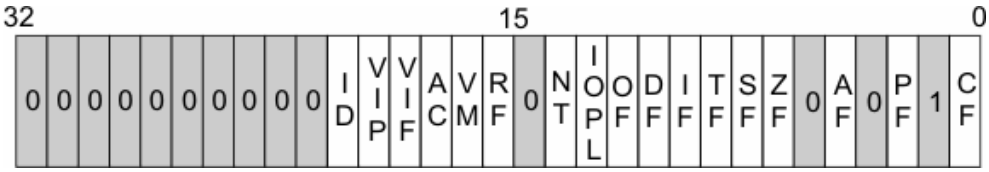

Рис. 1.7. Содержимое регистра EFLAGS

Таблица 1.1. *Описание флагов*

Флаг	Название	Номер бита в eflags	Назначение
CF	Флаг переноса (Carry Flag)	0	Устанавливается, если был перенос из старшего бита при операции сложения или заем бита при операции вычитания.
PF	Флаг четности (паритета) (Parity Flag)	2	Устанавливается, если младший байт содержит четное число единиц, и сбрасывается, если в младшем байте количество единиц нечетное.
AF	Флаг вспомогательного переноса (Auxiliary carry Flag)	4	Применяется для работы с BCD-числами. Устанавливается, если в результате операции сложения был произведен перенос из бита 3 в бит 4, или при операции вычитания – заем из бита 4 в бит 3.
ZF	Флаг нуля (Zero Flag)	6	Устанавливается, если результат операции равен 0 (все биты результата равны нулю). Сбрасывается, если результат не нулевой
SF	Флаг знака (Sign Flag)	7	Равен старшему биту результата последней команды
TF	Флаг ловушки (трассировки) (Trace Flag)	8	Используется отладчиками в реальном режиме. Установка в 1 этого флага приводит к тому, что после каждой команды вызывается прерывание с номером 1.
IF	Флаг прерываний (Interrupt enable Flag)	9	Сброс этого флага в 0 приводит к тому, что микропроцессор перестает воспринимать прерывания от внешних устройств.
DF	Флаг направления (Directory Flag)	10	Учитывается в операциях со строками: 0 – строки обрабатываются от младших адресов к старшим; 1 – строки обрабатываются от старших к младшим. Устанавливается с помощью команды std, очищается с помощью команды cld

Таблица 1.1. *(окончание)*

OF	Флаг переполнения (Overflow Flag)	11	Устанавливается, если результат операции над числом со знаком вышел за допустимые пределы
IOPL	Уровень привилегий ввода-вывода (Input/Output Privilege Level)	12,13	Используется в защищенном режиме работы микропроцессора. Определяет, какой привилегией должен обладать код, чтобы ему было разрешено выполнять команды ввода-вывода и другие привилегированные команды.
NT	Флаг вложенности задачи (Nested Task)	14	Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую
RF	Флаг возобновления (Resume Flag)	16	Служит для временного выключения обработки исключительных ситуаций отладки для того, чтобы команда, вызвавшая такую ситуацию, могла быть перезапущена и не стала бы причиной новой исключительной ситуации. Отладчик устанавливает этот флаг с помощью команды iretd при возврате в прерванную программу
VM	Флаг виртуального 8086 (Virtual 8086 Mode)	17	Признак работы микропроцессора в режиме виртуального 8086 (V86): 1 – процессор работает в режиме V86; 0 – процессор работает в реальном или защищенном режиме
AC	Флаг контроля прерывания (Alignment Check)	18	Предназначен для контроля выравнивания при обращениях к памяти. Используется совместно с битом AM в системном регистре CR0. К примеру, микропроцессор Pentium разрешает размещать команды и данные с любого адреса. Если этот флаг установлен, то при обращении к невыровненному операнду будет вызываться исключение.
VIF	Флаг виртуального прерывания (Virtual interrupt flag)	19	Является виртуальным подобием флага IF и используется совместно с флагом VIP в защищенном режиме (только для Pentium и выше)
VIP	Ожидание виртуального прерывания (Virtual interrupt pending flag)	20	Указывает процессору, что произошло аппаратное прерывание. Флаги VIF и VIP используются совместно в многозадачных средах для того, чтобы каждая задача имела собственный виртуальный образ флага IF (только для Pentium и выше)
ID	Флаг доступности команды идентификации (Identification flag)	21	Служит для проверки доступности команды CPUID. Если в программе можно установить и сбросить флаг ID, то данный процессор поддерживает команду CPUID, предоставляющую программисту информацию о продавце, модели и поколении данного процессора (только для Pentium и выше)

1.5.2.2.Системные регистры

Эти регистры используются для обеспечения работы защищенного режима микропроцессора, поэтому редко используются программистами (рис. 1.8). К системным регистрам относят: четыре регистра системных адресов (GDTR, IDTR,

TR, LDTR), пять регистров управления (CR0 – CR4) и восемь регистров отладки (DR0 – DR7).

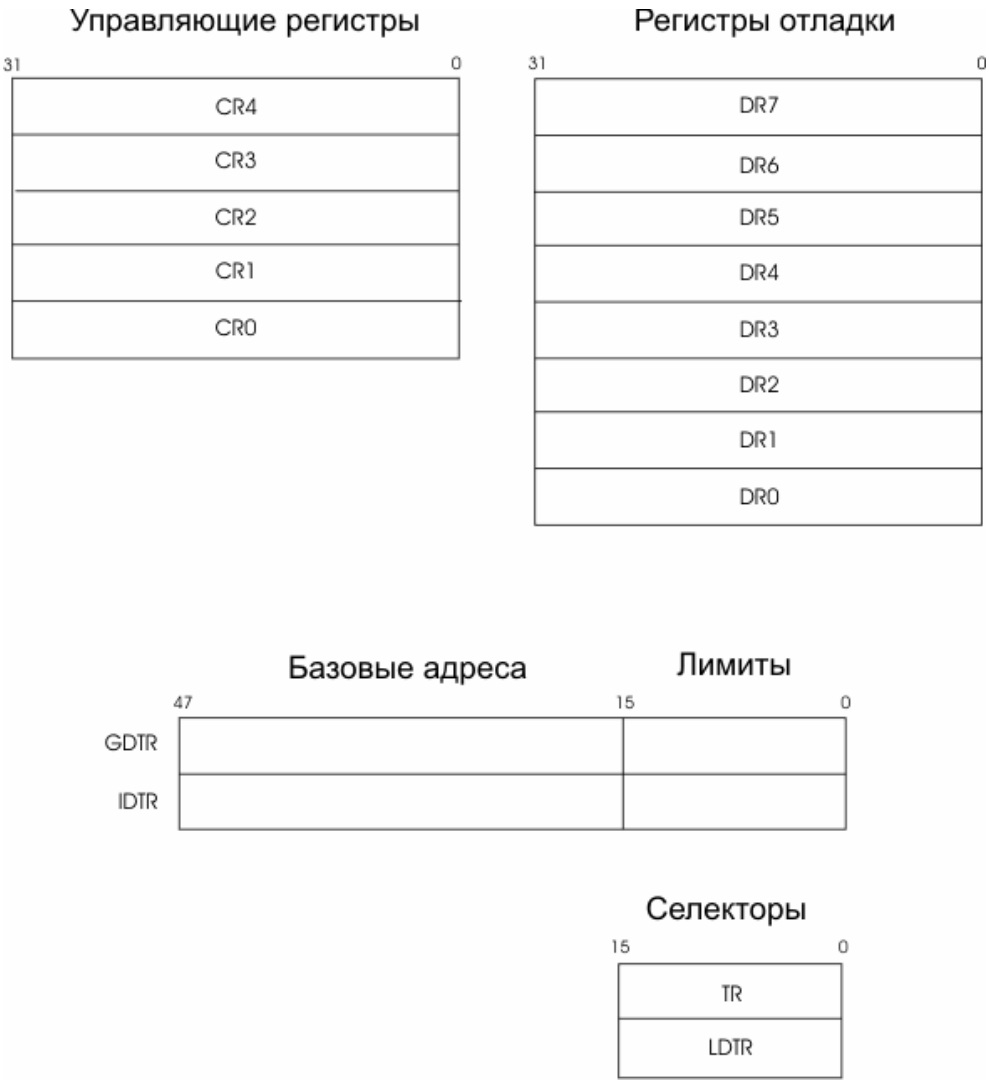

Рис. 1.8. Системные регистры

1.5.2.3. Регистры FPU и MMX

Регистры FPU (Floating Point Unit — блок чисел с плавающей запятой) предназначены для ускорения операций с числами с плавающей запятой (рис. 1.9). В первых поколениях процессоров эти регистры располагались в отдельной микросхеме, которая называлась *сопроцессор* на материнской плате. Для соответствующего поколения процессора был свой сопроцессор: 8087, 80287, 80387, 80487. Начиная с процессора 80486DX, сопроцессор располагается на одном кристалле с центральным процессором. В разных поколениях процессоров сопроцессор, называли, по-разному FPU или NPX (Numeric Processor eXtention — числовое расширение процессора), однако первое название получило наибольшее распространение.

В блок FPU входят пять вспомогательных регистров:

- регистр состояния SWR (Status Word Register)
- регистр управления CWR (Control Word Register)
- регистр тегов TWR (Tags Word Register)
- регистр-указатель команд IPR (Instruction Point Register)
- регистр-указатель данных DPR (Data Point Register)

Регистры MMX (MultiMedia eXtensions — мультимедийные расширения) появились в пятом поколении процессоров Intel (рис. 1.9). MMX ускоряют работу с мультимедийными приложениями. Это достигается за счет одновременной обработки нескольких элементов данных за одну инструкцию — так называемая технология SIMD (Single Instruction — Multiple Data).

Рис. 1.9. Регистры FPU и MMX

Регистры MMX и FPU/NPX являются одними и теми же регистрами сопроцессора, просто в программе при необходимости программист явно указывает, желает он использовать эти регистры для работы с мультимедийными приложениями (MMX) или для работы с числами с плавающей запятой (FPU/NPX).

1.5.2.4. Регистры XMM (расширение SSE/SSE2)

Впервые расширение SSE (Streaming SIMD Extensions — потоковые SIMD-расширения) появились в процессоре Pentium III. Расширение предназначено для ускорения работы с 2D/3D, видео-, аудио- и другими видами потоковых данных. Только в отличие от MMX, которое ограничивается целочисленной арифметикой и логикой, расширение SSE работает с числами с плавающей точкой. Расширение вводит 8 новых независимых 128-битных регистров данных: XMM0-XMM7 и регистр состояния/управления MXCSR (рис. 1.10).

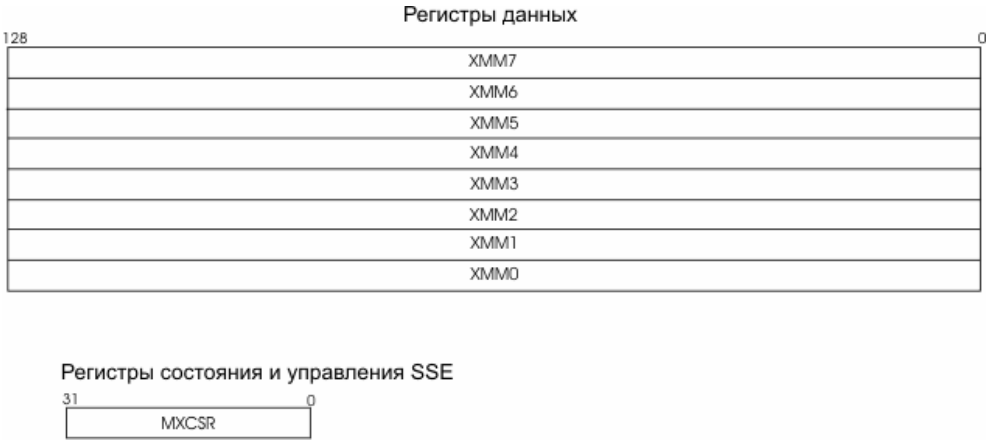

Рис. 1.10. Регистры XMM (расширение SSE/SSE2)

В процессоре Pentium 4 появилось очередное расширение — SSE2. Это расширение не добавило новые регистры, но появились новые инструкции для работы с данными.

1.6. Память

Надеюсь, уважаемый читатель знает, сколько оперативной памяти установлено в его компьютере. Думаю не меньше 1 Гбайт. Если меньше, то советую прикупить, т. к. это мало по нынешним меркам (память в наше время стоит сравнительно дешево).

С точки зрения программиста память состоит из отдельных ячеек размером в байт (8 бит). Точнее память состоит из битов, но программист может оперировать только отдельными байтами. Как уже говорилось, если программе нужно изменить значение только одного бита, то она все равно должна считать целый байт, содержащий этот бит.

Т. к. ячеек памяти, в отличие от процессорных регистров, огромное количество, то они не имеют названий как регистры процессора, а имеют просто уникальные числовые адреса, называемые физическими. Таким образом, память это просто огромный массив пронумерованных ячеек (нумерация начинается с нуля).

Вся память делится на оперативную (ОЗУ) (по-английски RAM (Random Access Memory) — устройство с произвольным доступом) и постоянную память (ПЗУ) (ROM (Read Only Memory) — память только для чтения). Если в ОЗУ можно как записывать, так и считывать информацию, то из ПЗУ ее можно только считывать. В ПЗУ расположена BIOS и программа начальной загрузки компьютера (POST). Постоянная и оперативная память находятся в едином пространстве адресов.

Важно помнить, что перед выполнением *любая* программа должна быть загружена в ОЗУ, только после этого процессор начинает последовательно считывать из нее и выполнять команды. Жесткие диски, дискеты, CD/DVD и прочие носители информации хранят файлы, которые будут выполнены только после того как будут загружены в память, причем образ на носителе информации не всегда соответствует тому образу, который будет перенесен в память. Переносом программы с носителя в память (и обратно, если необходимо) занимается операционная система.

Загруженная в память программа всегда отводит под свои нужды отдельный участок памяти, который называется *стеком*. Стек работает особым образом — данные в него помещаются и извлекаются по принципу LIFO (Last In First Out — "последним вошел – первым вышел"). Стек можно представить в виде стопки листов бумаги (это, кстати, одно из значений английского слова stack) — листы, которые мы положили в стопку последними, сможем забрать первыми, иначе говоря, можем класть и забирать листы только с вершины стопки. Как вы помните, существуют специальные регистры, отведенные для работы со стеком это: SS, ESP, EBP. Далее в книге мы рассмотрим подробно работу стека. В системе Intel дно стека всегда расположено в больших адресах памяти, т. е. стек заполняется от максимально возможного адреса к меньшим. Программа и данные заполняют память, начиная с малых адресов памяти к большим. Между стеком и программой с данными существует промежуток из неиспользуемых адресов памяти.

При рассмотрении сегментных регистров (разд. 1.5.2.1.2) я уже говорил, что в реальном режиме процессор делит пространство памяти на сегменты по 64 Кбайт (сегментированная модель памяти), в защищенном режиме процессор предоставляет несколько различных моделей памяти, но чаще всего используется самая простая плоская модель памяти (память представляется одним сплошным массивом байтов). Как выглядит образ памяти программы в реальном режиме показано на рис. 2.1 и на рис. 2.2, а плоская модель памяти показана на рис. 1.11.

Вас, возможно, интересует, зачем фирме Intel понадобилось в реальном режиме делить память на 64 Кбайт сегменты? Разумеется, это была не просто прихоть фирмы Intel. Процессоры первого поколения были 16-разрядными, а, следовательно, максимальный размер, который они могли адресовать, составлял всего 2^{16}=64 Кбайт. Понятно, что это очень мало, поэтому Intel стала искать способы расширения доступного адресного пространства. Конечно, самый простой способ — это увеличить разрядность процессора (что и было сделано в последующих поколениях), но в первых поколениях процессоров это не позволяла сделать технология, ограничивающая количество элементов на чипе. Поэтому фирма Intel решила использовать специальный встроенный диспетчер памяти, для управления которым были введены известные нам уже сегментные регистры: CS указывал на область ОЗУ в котором располагался код программы, регистр DS отвечал за данные, SS определял расположение стека. А адрес ячейки внутри сегмента стал представлять собой совокупность двух слов, записываемых в программах в виде `SSSSh:OOOOh`, где `SSSSh` — адрес сегмента, а `OOOOh` — относительный адрес (называемый также эффективным), или *смещение*, который используется для доступа к ячейке внутри сегмента. Адрес, состоящий из сегмента и смещения, называют обычно *логическим* или *виртуальным* адресом.

Однако в процессорах первого поколения использовалась 20-разрядная шина адреса, по которой передать значение адреса состоящего из двух слов (32 бита) было невозможно. Поэтому для преобразования 32-х разрядного адреса в 20-разрядный адрес для передачи по шине адреса Intel ввела следующий аппаратный алгоритм: значение сегментного регистра умножается на 16 (10h) или (что то же самое) сдвигается на 4 разряда влево и складывается со значением смещения, в результате получается 20-битный адрес. Например, если 32-разрядный адрес DS:BX, где DS=1234h, BX=5678h, то значение сегментного регистра, умноженное на 16 будет равно 12340h, а физический адрес 12340h+5678h=179B8h.

Таким образом, под физическим адресом понимается адрес памяти, выдаваемый на шину адреса микропроцессора. Другие названия этого адреса — *абсолютный адрес*, *линейный адрес* (однако в защищенном режиме физический и линейный адреса — это не одно и то же, об этом ниже). Так как физический адрес имеет размерность 20 бит, то максимальное пространство памяти, которое может использовать программа в реальном режиме равно 2^{20}=1 Мбайт. Еще раз отмечаю, что преобразование в физический адрес выполняется на аппаратном уровне, поэтому вручную это делать от вас не потребуется, но знать, как в реальном режиме вычисляется физический адрес полезно.

Конечно, в наше время размер 1 Мбайт памяти, который предоставлен программе в реальном режиме, выглядит смешно. Но когда-то это казалось очень много. Ограничение памяти в 1 Мбайт в реальном режиме, также как и в режиме V86 сохранилось до сих пор — об этом вам следует помнить.

В последующих поколениях процессорах разрядность шины адреса увеличивалась, например в Pentium 4 она составляет 64 бита, но все равно в реальном режиме не зависимо от поколения процессора задействуются только 20 линий, а остальные линии шины адреса в этом режиме просто недоступны. Начиная с процессора 80386, в качестве смещения стало возможно использовать 32-х разрядный адрес.

Таким образом, имея 16-разрядные регистры, удалось увеличить адресное пространство до 1 Мбайт, а данную технологию назвали *сегментацией памяти*. Конечно, тогда это решение казалось удачным, но с появлением защищенного режима в 32-разрядных процессорах фирма Intel перешла к плоской модели памяти, а сегментную модель памяти пришлось сохранить для обеспечения совместимости с программным обеспечением, созданным под реальный режим работы процессора.

В защищенном режиме появилось еще 4 модели памяти (эти модели памяти недоступны в реальном режиме):

- **плоская, или линейная модель памяти (flat memory model)** — вся память представляет собой непрерывную линейную последовательность байт (рис. 1.11). Диапазон адресов в этой модели находится в пределах от 0 до 2^{32}-1 (4 Гбайт). Программный код, данные и стек располагаются в этом пространстве адресов.

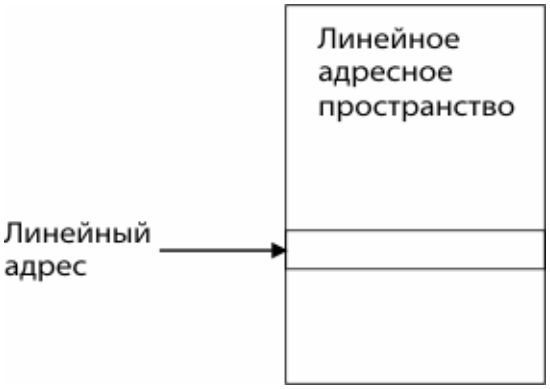

Рис. 1.11. Плоская модель памяти

- **сегментированная модель памяти (segmented memory model)** — подобно тому, как это делается в реальном режиме, в защищенном режиме память может делиться на отдельные пространства адресов, которые называют сегментами. При этом программный код, данные и стек размещаются в отдельных сегментах памяти. Программы в 32-разрядном режиме могут использовать до 16383 сегментов разного размера, каждый из которых может иметь размер 2^{32} байт (4 Гбайт). Однако в отличие от реального режима, преобразование логических адресов в физические в сегментированной модели памяти защищенного режима выполняется значительно сложнее. По-прежнему логический адрес формируется при помощи сегментных регистров и регистров, в которых хранятся смещения. Однако сегментные регистры теперь хранят не сегментный адрес, а так называемый *селектор* (рис. 1.12).

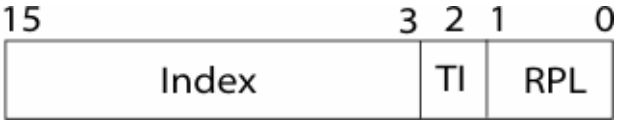

Рис. 1.12. Формат селектора

Он также содержит 16 бит, но теперь имеет более сложную структуру:

- **Index** — индекс в таблице дескрипторов (его длина 13 бит, следовательно в таблице содержится не более 2^{13}=8192 дескрипторов).
- **TI** — если бит установлен, то это селектор в LDT, сброшен в GDT.
- **RPL** — уровень привилегий запроса.

Индекс селектора (13 бит) указывает на дескриптор в таблице, называемой дескрипторной.

В сегментированной модели памяти защищенного режима используется две дескрипторные таблицы: глобальная (GDT) и локальная (LDT). Тип используемой таблицы определяется битом TI селектора. Таблицы — это просто массивы из дескрипторов. Адреса этих массивов хранятся в системных регистрах: GDTR и LDTR, соответственно (см. рис. 1.8).

Селекторы текущих сегментов кода, данных и стека хранятся в регистрах CS, DS и SS соответственно. Таким образом, логический адрес формируется из селектора сегмента и смещения внутри сегмента. Исходя из всего вышеперечисленного

схема адресации сегментированной модели памяти защищенного режима будет выглядеть так как показано на рис. 1.13.

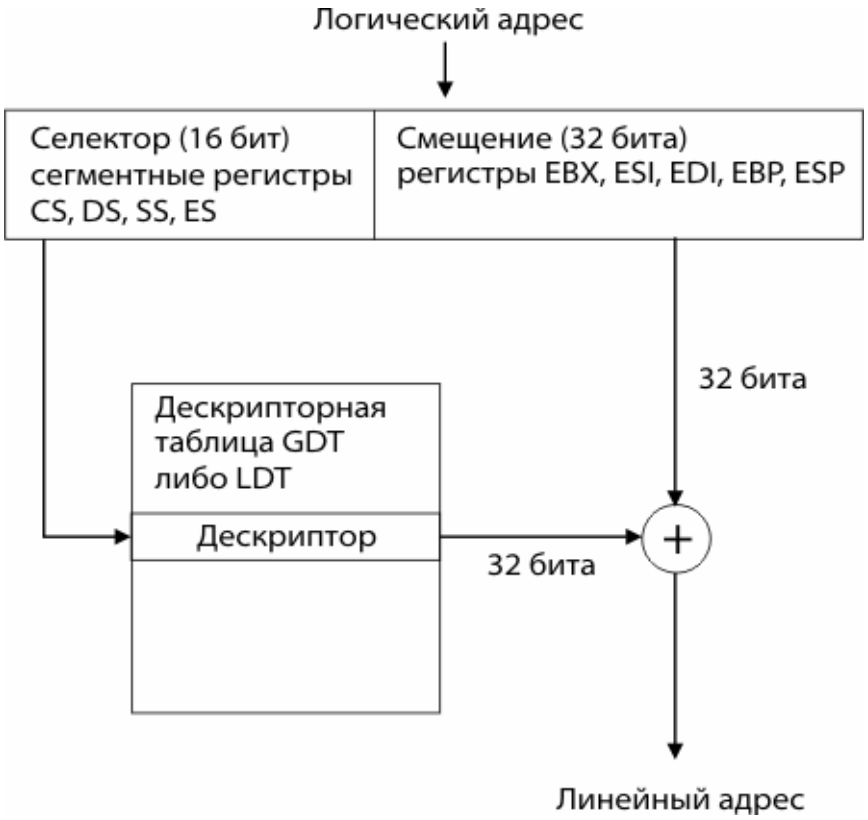

Рис. 1.13. Схема преобразования логического адреса в линейный в защищенном режиме адресации

- **страничная модель памяти (paging)** — является надстройкой над сегментированной или плоской моделью памяти. В этом режиме память рассматривается как совокупность блоков фиксированного размера (страниц) размером 4 Кбайт. Начиная с 5-го поколения процессоров, появилась возможность увеличения размера страницы до 4 Мбайт. Страничная модель памяти предназначена для организации виртуальной памяти. Благодаря виртуальной памяти программы могут использовать для работы объем памяти больший, чем объем физической памяти. Суть виртуальной памяти заключается в том, что страницы могут выгружаться из физической оперативной памяти на диск (в файл обмена, файл подкачки или swap-файл) и по мере необходимости подкачиваться с него обратно. Через страничное преобразование i386 может адресовать до 4 Гбайт физической памяти и до 64 Тбайт виртуальной памяти. Виртуальная память включается в настройках операционной системы. Разбиение на страницы выполняется на аппаратном уровне и программисту знать подробностей, как правило, не требуется. На программном уровне страничный механизм включается установкой специального бита (PG) в регистре CR0 при помощи привилегированной команды.
- **модель памяти в режиме V86.** С точки зрения программиста эта модель памяти работает точно также как в обычном реальном режиме. Т. е. память делится на сегменты по 64 Кбайт, ячейки внутри которых адресуются с помощью двух слов, записываемых в виде `СЕГМЕНТ:СМЕЩЕНИЕ`, максимальная адресуемая память 1 Мбайт и пр. Однако в режиме V86 выполняются все проверки защиты защищенного режима, из-за чего в некоторых случаях не будут работать некоторые инструкции. Особенно это касается инструкций ввода-вывода IN, OUT, (REP) INS, (REP) OUTS и инструкций обработки прерываний: INT n, PUSFF, POPF, STI, CLI и IRET. Эти инструкции мы еще будем изучать на следующих днях.

Если включен режим страничной переадресации, то физический адрес не совпадает с линейным. В этом случае трансляцию линейных адресов в физические выполняет специальный блок страничной переадресации, впрочем, нам знать все тонкости этих преобразований ни к чему, но если вам интересно, то советую обратиться, например к [1].

1.7. Порты ввода-вывода

Передача данных и управление работой всеми внешними устройствами происходит через порты ввода-вывода. К внешним устройствам относятся: устройства ввода-вывода (клавиатура, мышь, дисплей, принтер, акустические системы, плоттер и пр.), устройства хранения информации (дисковод, жесткий диск, CD-ROM и пр.), коммуникационные устройства (сетевые адаптеры, последовательные, параллельные и прочие порты) и пр. То есть фактически все устройства компьютерной системы по отношению к памяти и центральному процессору являются внешними, причем они могут находиться даже на одной плате (материнской плате) с процессором.

Не нужно только путать параллельный порт (LPT) и последовательный порт (COM) с портами ввода-вывода. LPT и COM это электронные схемы, которые являются коммуникационными устройствами, и управляются они точно также через порты ввода-вывода.

Порты ввода-вывода это микросхемы, через которые центральный процессор передает внешним устройствам команды и получает от внешних устройств данные и информацию о состоянии (статусе) устройства. В свою очередь большинство внешних устройств подключаются к портам ввода-вывода через свои контроллеры (специальные микросхемы) или адаптеры. Порты ввода-вывода имеют свои уникальные адреса (отличные от адресов ячеек памяти) по которым центральный процессор обращается к ним. Например, порт ввода-вывода 3F8h не имеет ничего общего с адресом памяти 3F8h.

Если максимальный объем адресного пространства памяти составляет 1 Мбайт в реальном режиме, а в защищенном режиме 4 Гбайт, то адресное пространство портов гораздо меньше — его размер составляет всего 64 Кбайт (диапазон номеров от 0 до 65635). Порты ввода-вывода можно рассматривать как вместе, так и отдельно: порты ввода и порты вывода. За некоторыми стандартными устройствами закреплены определенные адреса портов ввода-вывода (см. табл. 1.2). На различных материнских платах диапазоны адресов могут немного отличаться от представленных в таблице, хотя большинство адресов портов ввода-вывода стандартных устройств не меняются, начиная с первых поколений процессоров.

Таблица 1.2. *Распределение адресов портов ввода-вывода*

Диапазон	**Использование**
000 – 01Fh	Контроллер DMA №1
020h – 03Fh	Контроллер прерываний №1
040h – 05Fh	Таймер
060h – 06Fh	Контроллер клавиатуры
070h – 07Fh	RTC, CMOS, RAM
080h – 09Fh	Порты DMA (регистры страниц)
0A0h – 0AFh	Контроллер прерываний №2
0C0h – 0DFh	Контроллер DMA №2
0E0h – 0EFh	Зарезервировано
0F0h – 0FFh	Математический сопроцессор
170h – 177h	Жесткий диск (вторичный)
1F0h – 1FFh	Жесткий диск
200h – 207h	Игровой адаптер

Таблица 1.2. *(окончание)*

278h – 27Fh	Параллельный порт №2
2C0h – 2DFh	Адаптер EGA №2
2F8h – 2FFh	Последовательный порт №2
300h – 31Fh	Платы прототипов
320h – 36Fh	Зарезервировано
370h – 377h	Контроллер дисковода (вторичный)
378h – 37Fh	Параллельный порт №1
3B0h – 3DFh	Адаптер VGA
3C0h – 3CFh	Адаптер EGA №1
3D0h – 3DFh	Адаптер CGA и EGA
3F0h – 3F7h	Контроллер дисковода
3F8h – 3FFh	Последовательный порт №1

1.8. Шины

Как уже говорилось в самом начале этого дня (см. разд. 1.1), все узлы компьютерной системы соединены между собой при помощи магистрали, которая состоит из трех шин:

- ШД — шина данных (DATA bus);
- ША — шина адреса (ADDR bus);
- ШУ — шина управления (CONTROL bus).

Все данные перемещаются по шине данных, а адреса перемещаются по шине адреса. Шины также имеют разрядность. Разрядность шины означает, сколько одновременно (параллельно) бит данных может перемещаться по ней (понятно, что чем больше, тем лучше). Шестнадцатиразрядная шина данных может за раз передавать до двух байтов, 32-разрядная шина передает до четырех байт, 64-разрядная до восьми.

По шине адреса перемещаются адреса ячеек памяти или адреса портов ввода-вывода, к которым в данный момент обращается процессор.

По шине управления передаются различные управляющие сигналы от процессора на внешние устройства и обратно, такие как: сигнал записи, сигнал чтения, сигнал инициализации памяти и портов ввода-вывода и пр.

Программисту на ассемблере не обязательно знать, что, куда, в какой момент и в каком порядке передается по этим трем шинам, так как это все происходит автоматически и скрытно от программиста.

ДЕНЬ 2

Основы программирования на ассемблере

Каждая программа выполняется под управлением операционной системы (разумеется, если программа сама не является операционной системой). Операционная система выделяет для программы области памяти, загружает программу в память, передает ей управление, обеспечивает взаимодействие с другими программами и пр. Различные операционные системы это делают по-разному, поэтому программа, написанная для ОС Windows, не будет работать в ОС MS-DOS, а тем более в ОС Linux. Конечно, можно написать программу, которая не будет рассчитана ни на одну известную операционную систему, но в этом случае она сама должна выполнять основные функции операционной системы, т. е. по сути, должна являться мини операционной системой.

Мы начнем изучать ассемблер под операционной системой MS-DOS (Microsoft Disk Operating System — дисковая операционная система фирмы Microsoft). Это одна из самых древних и простых операционных систем, которая использовалась еще на первых IBM PC. Не волнуйтесь, бежать искать дистрибутив с MS-DOS вам не потребуется. Если у вас на компьютере установлена Windows, то, значит, установлена и MS-DOS. MS-DOS является неотъемлемой частью любой из версий Windows. Последней отдельно распространяемой версией MS-DOS была версия 6.22, выпущенная в мае 1994 года.

Зачем нужно изучать ассемблер под устаревшей операционной системой? Во-первых, в отличие от Windows, которая является операционной системой защищенного режима, MS-DOS является операционной системой реального режима. Поэтому, программируя под MS-DOS, мы изучим программирование в реальном режиме. Во-вторых, и самое важное, начинать программировать под MS-DOS проще в том плане, что не нужно изучать функции WinAPI и построение графических приложений, которые являются основными в Windows. Программирование в Windows осуществляется только через API функции, предоставляемые этой системой. Даже для того чтобы вывести что-нибудь в консольное окно в Windows нужно задействовать соответствующие API-функции, т. е. Windows скрывает от программиста работу с устройствами компьютера. Поэтому программирование на ассемблере под MS-DOS позволяет лучше познакомиться с возможностями ассемблера, т. к. MS-DOS дает возможность напрямую работать с устройствами компьютера (процессором, памятью, портами ввода/вывода). К сожалению MS-DOS, работающая внутри Windows, в отличие от чистой MS-DOS, существенно ограничивает работу DOS-приложений. Надо заметить, что в первых версиях Windows построенных на не NT-ядре (Windows 3.1, Windows 95, Windows 98, Windows Millennium) никаких ограничений для DOS-приложений не существовало.

Возможно, любознательный читатель сейчас задался вопросом: каким образом MS-DOS, которая является ОС реального режима может работать в Windows, которая является ОС защищенного режима? На самом деле в Windows работает не ОС MS-DOS в чистом виде, а просто виртуальная машина, которая эмулирует работу MS-DOS (virtual DOS-machine, VDM). Вспомните, на первом дне мы уже упоминали о VDM (см. разд. 1.5.1. "Режимы работы процессора").

VDM — это обычное приложение Windows, которое создает виртуальный компьютер, исполняющий MS-DOS. Благодаря VDM вы можете открыть сразу множество окон с исполняющимися независимо друг от друга DOS-программами (в реальной MS-DOS такое было невозможно). Программировать на ассемблере в VDM

безопаснее и удобнее тем, что в случае фатальной ошибки в программе обычно достаточно просто закрыть окно с виртуальной машиной и открыть новое. При программировании же в реальной MS-DOS (не в VDM) нередко приходилось перезагружать весь компьютер или даже переустанавливать систему, впрочем, этим грешили и первые версии Windows созданные не на NT-ядре.

Перед запуском на исполнение любого файла, Windows автоматически определяет, является ли данный файл DOS-файлом или нет, и если является, то выполняет его в VDM. Windows это делает по специальному служебному заголовку исполняемого файла, — этот заголовок вставляет компилятор или транслятор в начало каждого исполняемого файла при его создании.

2.1. Какой ассемблер выбрать

Но прежде чем начать писать программы на ассемблере мы должны выбрать, какой ассемблер будем использовать. Наиболее известными и распространенными в наше время ассемблерами являются:

- **MASM (Macro Assembler)** — ассемблер от фирмы Microsoft. Сначала MASM был коммерческим продуктом, но затем Microsoft стала распространять его в составе бесплатного DDK (Device Driver Kit — пакет разработчика драйверов), а также в составе Visual C++. Хотя Microsoft официально не объявляла MASM бесплатным продуктом, но фактически этот ассемблер в настоящее время программисты могут использовать бесплатно. MASM является основным ассемблером под Windows, т. к. большинство примеров на ассемблере, в том числе от самой Microsoft, распространяются именно на MASM. Поэтому изучать никакие другие ассемблеры под DOS/Windows в общем-то не имеет особого смысла, к тому же второй по популярности ассемблер в DOS/Windows – TASM, способен компилировать программы написанные на MASM. Microsoft продолжает совершенствовать свой продукт. Самая последняя версия MASM 8.0, включенная в состав Visual C++ 2005, имеет поддержку всех новых инструкций процессора (MMX, SSE, SSE2, SSE3, 3DNow!) и поддержку 64-разрядного режима.
- **MASM32** — ассемблер созданный на основе MASM независимым программистом Стивом Хатчессоном (Steve Hutchesson или просто hutch) специально для программистов под Windows. На самом деле это тот же самый MASM от Microsoft (MASM32 версии 10 основан на MASM версии 6.14.8444), только в один пакет с ним включено множество библиотек облегчающих программирование, документации, примеров программ, полезных утилит, в том числе простенькое IDE (Integrated Development Environment — интегрированная среда разработки) и пр. Документация от MASM32, перевод которой существует в том числе на русском языке (можно найти на **http://www.wasm.ru**), является отличным введением в программирование под Windows. MASM32 совершенно бесплатен и продолжает активно развиваться. Официальный сайт, где можно скачать пакет MASM32: **http://www.masm32.com**.
- **TASM (Turbo Assembler)** — ассемблер фирмы Borland, который во времена MS-DOS был очень популярен. Особенностью TASM является то, что он способен работать в режиме полностью совместимом с ассемблером MASM, а также в своем собственном режиме IDEAL (использует свой синтаксис). К сожалению, фирма Borland прекратила разработку и распространение своего ассемблера.

  Последняя версия транслятора TASM 5.0, поддерживает команды только до 80486 процессора включительно. Отдельно был выпушен патч, обновляющий TASM до версии 5.3 и позволяющий работать с инструкциями Pentium MMX.

  Теперь TASM можно найти только в интернете или на пиратских дисках.

  В интернете (например на **http://www.wasm.ru**) можно найти пакет TASM 5+ созданный энтузиастом по кличке !tE специально для программистов под Windows, включающий в себя кроме ассемблера TASM версии 5.3, документацию, несколько заголовочных файлов под Windows и пару демонстрационных примеров. Только не перепутайте этот пакет с TASM32 от фирмы Squak Valley Software — это совершенно независимый кроссассемблер[3],

[3] Кроссассемблером называют ассемблер, запускающийся на одной системе, а производящий код для другой.

ориентированный на специфические процессоры 6502, 6800/6801/68HC11, 6805, TMS32010, TMS320C25, TMS7000, 8048, 8051, 8080/8085, Z80, 8096/80C196KC.

Стоит еще упомянуть, что наш соотечественник Половников Степан создал бесплатный ассемблер для DOS и Windows под названием Lazy Assembler (**http://lzasm.hotbox.ru**), совместимый с режимом IDEAL TASM и поддерживающий все современные команды процессоров: MMX, SSE, SSE2, SSE3 (PNI), SSE4 (MNI), 3DNow!Pro.

- **NASM (Netwide Assembler)** — бесплатный переносимый ассемблер с открытым исходным кодом (open source) под лицензией GNU Lesser General Public License (LGPL). Существуют версии NASM под операционные системы DOS, Windows, Unix, OS/2 и др., а также под 32- и 64-разрядные процессоры, т. е. изучив единожды синтаксис этого языка можно не переучиваясь программировать практически под любой платформой. NASM хорошо документирован (в т. ч. можно найти переводы документации на русский язык) и продолжает совершенствоваться. Однако под Windows этот ассемблер почти не используется, т. к. исторически монополия здесь принадлежит ассемблеру MASM. К тому же многие программисты находят не очень удобным синтаксис NASM. Сайт ассемблера NASM: **http://nasm.sourceforge.net**.
- **FASM (Flat assembler)** — очень популярный в последнее время ассемблер, созданный поляком Томашом Гриштаром (Tomasz Grysztar). FASM бесплатен и распространяется с исходным кодом (open source) под лицензией GNU Public Licence (GPL). На FASM были полностью написаны миниатюрные операционные системы MenuetOS и KolibriOS. Примечательно, что сам FASM был написан на себе самом. FASM имеет уникальный синтаксис с очень развитым макроязыком и поддерживает все современные инструкции 32- и 64-разрядных процессоров (MMX, SSE, SSE2, SSE3, 3DNow!). Существуют версии FASM под операционные системы DOS, Windows, Linux и версии UNIX, которые имеют поддержку ELF-формат файлов и библиотеку Си. Официальная страница FASM: **http://flatassembler.net**.
- **AS** — стандартный ассемблер практически во всех разновидностях UNIX, в том числе Linux и BSD. Свободная версия этого ассемблера называется gas (GNU assembler). Синтаксис AS (GAS) в корне отличается от синтаксиса предыдущих ассемблеров.

Существует еще огромное число менее известных ассемблеров, созданных как большими фирмами, так и программистами-одиночками, например WASM, Pass32, YASM и многие другие.

Исторически в мире ассемблеров сложилось два основных типа синтаксисов: Intel-синтаксис и AT&T-синтаксис. Первый был предложен фирмой Intel и широко использовался в ассемблерах под MS-DOS. Синтаксис AT&T с самого начала стал использоваться в ассемблерах под UNIX системы.

Сейчас все существующие ассемблеры придерживаются с некоторыми оговорками либо синтаксиса Intel, либо синтаксиса AT&T. Ассемблеры MASM, MASM32, TASM, NASM, FASM используют Intel-синтаксис. Ассемблеры AS и gas используют синтаксис AT&T.

По своему опыту скажу, что если хорошо научиться программировать в Intel-синтаксисе, то перейти потом на синтаксис AT&T будет очень легко.

Мы в этой книге будем использовать только Intel-синтаксис, но в приложении 3 я привел таблицу сравнения двух синтаксисов ассемблера, с примерами кода, чтобы вы имели общее представление об AT&T-синтаксисе.

При изучении программирования под ОС MS-DOS мы будем использовать только MASM, а при изучении программирования под ОС Windows — пакет MASM32.

2.2. Этапы создания программы

Надо сразу смириться с тем, что программировать на ассемблере придется без среды программирования (IDE), столь привычной в языках высокого уровня таких как Visual Basic, Delphi, Visual C++ и др. Правда существуют IDE и для ассемблера, созданные независимыми разработчиками, например VisualASM, TASMED, Turbo Assembler Shell и др., но их нечасто используют программисты на ассемблере.

Создание исполняемой программы на ассемблере можно разбить на следующие этапы:

1. ВВОД ИСХОДНОГО ТЕКСТА. Пишется код на ассемблере в любом текстовом редакторе, например в стандартном Блокноте Windows (Notepad). Я лично люблю использовать встроенный редактор в файловом менеджере Far Manager, который вызывается клавишей <Shift>+<F4>. Скачать Far Manager можно с официального сайта **http://www.farmanager.com**. Жители xUSSR могут использовать Far Manager бесплатно.
2. ТРАНСЛЯЦИЯ. После того, как текст программы готов, он сохраняется в обычный текстовый файл с расширением .asm и передается через командную строку транслятору. Транслятор — это собственно и есть ассемблер, он переводит текст языка в машинный код (создает объектный файл .obj). Файл **masm.exe**, в пакете MASM, является транслятором. Параллельно с объектным файлом могут создаваться и другие вспомогательные файлы, например, файл листинга (.lst), предназначенный для отслеживания ошибок.
3. КОМПОНОВКА (ЛИНКОВКА). Полученный объектный файл, также через командную строку передается компоновщику (линковщику). И если не замечено ошибок, компоновщик выдает готовый исполняемый модуль (.exe). Компоновщик, как правило, стандартно входит в один пакет с ассемблером. Например, утилита **link.exe** в пакете MASM, это компоновщик.
4. ОТЛАДКА. Полученный исполняемый файл просматривается отладчиком на наличие логических ошибок. Этот этап является необязательным и обычно необходим только для больших программ. Существует огромное количество всевозможных отладчиков. Раздел 2.11 будет посвящен знакомству со стандартным DOS-отладчиком CodeView от Microsoft, который входит в пакет MASM (файл **cv.exe** в подкаталоге \BINR).

2.3. Структура программы

Программа на языке ассемблера состоит из строк, имеющих следующий общий вид:

```
[метка:] [команда или директива] [операнды] [;комментарий]
```

Все эти четыре поля необязательны, в программе вполне могут присутствовать и полностью пустые строки для выделения каких либо блоков кода. Квадратные скобки здесь показывают, что заключенные в них поля необязательны, но ни в коем случае их не нужно набирать при вводе программы. Так поле операнда заполняется только для тех команд, которым требуется операнды (не все команды процессора имеют операнды). Вот пример реальной строки из программы на ассемблере:

```
DownRight: add di,320          ; Наращивание координаты Y
```

Разберем эту строку на отдельные поля:

`DownRight:` — является меткой,

`add` — команда,

`di,320` — это два операнда команды ADD,

`; Наращивание координаты Y` — комментарий начинается от знака ";" и продолжается до конца строки.

2.3.1. Метка

Метка — это просто имя, которое обычно служит для ссылок на команду помеченной меткой из других мест программы.

Метка может состоять из любой комбинации букв латинского алфавита, а также цифр и символов _, $, @, ?, но цифра не может быть первым символом метки. Обратите внимание, метка не может содержать пробелы, поэтому, когда программистам нужно записать несколько слов в метке, то они их либо записывают

слитно, начиная каждое слово с большой буквы (например: `LeftDownCicl`), либо разделяют слова символом подчеркивания (например: `left_down_cicl`). Метка в программе должна быть уникальной, ранее не определявшейся, а за ней должно обязательно стоять двоеточие (:). Регистр символов по умолчанию не распознается, но это различие можно включить при ассемблировании в опциях командной строки транслятора. Если метка стоит одна в строке, то обычно считается, что такая метка метит следующую команду. Пример:

```
Start:
inc bx          ; Метка Start: метит команду inc bx
```

2.3.2. Команда или директива

Во втором поле может стоять либо команда процессора, которая транслируется в исполняемый код, либо директива, которая не транслируется в исполняемый код, но служит для различных указаний самому ассемблеру. Формат строки директивы в основном совпадает с форматом строки команды, но перед директивой не может находиться поле метки. Использование основных команд и директив в программах на ассемблере мы рассмотрим в этой книге.

2.3.3. Операнды

Директива или команда может иметь один, два или три оператора, либо не иметь ни одного. Для многих команд наличие операнда обязательно, для других — запрещено. Операнды отделяются друг от друга запятыми. Если команда имеет два операнда, то первый операнд называется *приемник*, а второй *источник*. Источник никогда не изменяется в процессе выполнения команды. Например, в команде `MOV AX,BX` приемником является операнд AX, а источником операнд BX. После выполнения этой команды значение в регистре AX изменится, а в BX нет.

Стоит отметить, что поле операндов, единственное из всех полей не может располагаться на отдельной строке, т. к. операнды являются неотъемлемой частью команд или директив. Например:

```
ADD AX, BX  ; правильное размещение операндов
ADD   ; Ошибка! Команда ADD не может существовать без операндов.
AX,BX ; Ошибка! Нельзя размещать операнды на отдельной строке.
```

2.3.4. Комментарий

Текст начинающийся с символа ";" и до конца строки является комментарием. Комментарий может состоять из любых символов, включая русские буквы и пробелы. При ассемблировании транслятор не анализирует комментарии и не включает их в машинный код. Комментарии предназначены исключительно для программиста, для пояснения смысла программы. К комментариям обычно относят и полностью пустые строки в программе.

Язык ассемблера допускает и многострочные комментарии, для этого служит директива COMMENT имеющая следующий синтаксис:

```
COMMENT маркер
текст
маркер
```

Где `маркер` — любой символ, который начинает комментарий. Этот же символ должен заканчивать многострочный комментарий.

`текст` — любой текст, который может размещаться на нескольких строчках.

Пример:

```
COMMENT * все эти
строки являются
комментарием *
```

Многострочные комментарии обычно используются, когда временно нужно исключить (закомментировать) некоторый ее фрагмент.

2.4. Некоторые важные директивы ассемблера

Мы уже изучили одну директиву COMMENT, сейчас познакомимся еще с несколькими директивами, наиболее часто используемыми в программах на ассемблере. Остальные директивы (равно как и команды ассемблера) мы будем изучать по ходу чтения книги.

2.4.1. Директивы определения данных

Директивы определения данных задают начальное значение переменной[4], устанавливают ее тип (байт, слово, вещественное число и т. п.) и ставят в соответствие имя, которое будет использоваться для обращения к данным. Директивы определения данных в общем виде выглядят следующим образом:

```
имя_переменной  d* значение
```

где d* — одна из нижеприведенных директив:

db— резервировать память для данных размером 1 байт;

dw — резервировать память для данных размером 2 байта (слово);

dd — резервировать память для данных размером 4 байта (двойное слово);

df — резервировать память для данных размером 6 байт;

dp — резервировать память для данных размером 6 байт;

dq — резервировать память для данных размером 8 байт;

dt — резервировать память для данных размером 10 байт.

Вместо точного значения можно указать знак вопроса (?), тогда переменная считается неинициализированной (не получает начального значения) и ее значение на момент запуска программы может оказаться абсолютно любым (случайным).

Примеры:

```
Name1 db 13
Name2 dw ?
Name3 db 1,2,3,4,5,6,7,8,9,0Ah,0Bh,0Ch,0Dh,0Eh,0Fh
            db 'A'
            db 10,20,30,40,50,60
float_name dd 2.7e5
```

Директива

```
str db 'Hello, World!'
```

эквивалентна следующей директиве:

```
str db 'H','e','l','l','o',' ','W','o','r','l','d','!'
```

В директивах определения данных возможно применение конструкции повторения DUP. Конструкция DUP определяет количество повторений операнда. Например, следующее определение:

```
db 0,0,0,0,0,0,0,0
```

эквивалентно

```
db 8dup(0)
```

Еще пример конструкции DUP:

```
dw 100 DUP(?)        ; сто неопределенных слов
```

[4] Переменной в программировании называют некоторое хранимое в памяти значение, которое может изменяться во время выполнения программы.

Более сложный пример:

```
name db 3 dup('abc',?,6)
```

эквивалентно:

```
name db 'abc',?,6, 'abc',?,6,'abc',?,6
```

Возможны вложенные конструкции DUP, например:

```
db 4 dup(3 dup(8)) ; двенадцать восьмерок
```

2.4.2. Директива эквивалентности

Директива эквивалентности присваивает имени значение операнда:

```
имя EQU операнд
```

В этой директиве обязательно должны присутствовать и имя и операнд, причем операнд только один. Директива EQU является аналогом команды `#define` препроцессора языка C. На этапе ассемблирования все имена, определенные с помощью EQU в программе будут заменены соответствующим операндом.

Примеры:

```
Namber EQU 3
Message EQU 'Hello, World!$'
R EQU AX
Var EQU 4[si]
```

Директива эквивалентности может размещаться в любом месте программы, как внутри сегмента, так и вне его.

2.4.3. Директива присваивания

Директива присваивания очень похожа на директиву эквивалентности, но операнд может быть только константа или выражение, имеющее смысл константы. Кроме того, имя указанное этой директивой может переопределяться в программе.

Примеры:

```
X=1
offset=X+10 ; offset = 11
X=0
```

2.4.4. Директивы задания набора допустимых команд

По умолчанию MASM использует набор команд процессора 8086 и выдает сообщение об ошибке, если используется команда в программе, которую этот устаревший процессор не поддерживает. Чтобы возможно было использовать команды, появившееся в более новых процессорах необходимо использовать следующие директивы:

- .8086 — разрешены только команды 8086 (используется по умолчанию);
- .186 — разрешены команды 80186;
- .286 и .286c — разрешены непривилегированные команды 80286;
- .286p — разрешены все команды 80286;
- .386 и .386c — разрешены непривилегированные команды 80386;
- .386p — разрешены все команды 80386;
- .486 и .486c — разрешены непривилегированные команды 80486;
- .486p — разрешены все команды 80486;
- .586 и .586c — разрешены непривилегированные команды P5 (Pentium);
- .586p — разрешены все команды P5 (Pentium);

- .686 — разрешены непривилегированные команды P6 (Pentium Pro, Pentium II);
- .686p — разрешены все команды P6 (Pentium Pro, Pentium II);
- .8087 — разрешены команды NPX 8087;
- .287 — разрешены команды NPX 80287;
- .387 — разрешены команды NPX 80387;
- .MMX — разрешены команды IA MM;
- .K3D — разрешены команды AMD 3D.

Сразу после указания любой из этих директив в программе, возможно использовать соответствующие команды процессора.

2.4.5. Упрощенные директивы определения сегмента

Вспомните при рассмотрении памяти, мы говорили, что в реальном режиме процессор делит память на сегменты. Программист должен указать в своей программе, к какому сегменту относится каждая из ее частей. Это можно сделать с помощью упрощенных директив определения сегмента, основными из которых являются:

- `.data` или `.DATA` — определяет начало или продолжение сегмента инициализированных данных, т. е. данных, которые имеют начальное значение, например:

```
.data
   var1 dw 13
   str1 db "Text string"
```

- `.data?` или `.DATA?` — определяет начало или продолжение сегмента данных, в котором располагаются неинициализированные данные. При рассмотрении директив определения данных мы говорили, что неинициализированные данные определяются при помощи оператора `?`. Например:

```
.data?
   var1 dw ?
   db 10 dup(?)
```

  Однако ничто не мешает вам располагать неинициализированные данные вместе с инициализированными данными в сегменте данных, определяемом с помощью директивы `.data`. Преимущество директивы `.data?` перед `.data` в том, что при ее использовании уменьшается размер исполняемого файла, т. к. неинициализированные данные не занимают места. В случае `.data` неинициализированные данные заполняются случайными значениями, а потому файл увеличивается на соответствующий размер.

- `.const` или `.CONST?` — определяет начало или продолжение сегмента данных, в котором определены константы, т. е. данные которые не изменяются в ходе работы программы. Разумеется, вам ничто не мешает определять константы в сегменте данных с помощью директивы `.data`, однако преимущество `.const` состоит в том, что такой сегмент получает атрибут "только для чтения" (read only) для лучшей защиты констант от модификации.
- `.stack` [размер] или `.STACK` [размер] — определяет начало или продолжение сегмента стека. Необязательный параметр *размер* задает размер стека. Если размер не указан, размер стека предполагается равным 1 Кбайт.
- `.code` [имя] или `.CODE` [имя] — определяет начало или продолжение сегмента кода. Если *имя* не указано, то по умолчанию используется имя `_TEXT` (для моделей памяти tiny, small, compact и flat) или имя_модуля`_TEXT` для моделей памяти medium, large и huge.

2.4.6. Директива указания модели памяти

Кроме определения сегментов, программист должен обязательно указать модель памяти с помощью директивы `.MODEL`, которая определяет, сколько и какие сегменты должны быть в программе.

Данная директива должна находиться перед любой из директив определения сегментов в программе и имеет следующий вид:

```
.model модель [,язык][,модификатор]
```

Обязательным параметром этой директивы является только модель, которая может принимать следующие значения:

- TINY имеет один сегмент памяти размером не более 64 Кбайт, в котором размещаются коды, данные и стек.
- SMALL может иметь до трех сегментов: один 64-килобайтный сегмент для кода, один сегмент для данных и один сегмент стека.
- COMPACT то же, что и SMALL, но может иметь несколько сегментов данных.
- MEDIUM то же, что и SMALL, но может иметь несколько сегментов для кода.
- LARGE имеет несколько сегментов как для кода, так и для данных, при одном сегменте стека.
- HUGE разрешает данным быть больше, чем 64 Кбайт, но в остальном полностью похожа на модель LARGE. Реализуется в защищенном режиме процессора.
- FLAT — вся память определяется как единый сегмент для кодов, данных и стека размером 4 Гбайт. Основная модель памяти, используемая в защищенном режиме процессора.

Как видите, всего существует 7 моделей памяти.

Язык — необязательный параметр, который может принимать значения C, PASCAL, BASIC, FORTRAN, SYSCALL и STDCALL. Этот параметр необходим для связи модулей на ассемблере с языками высокого уровня.

Модификатор — необязательный параметр, который может принимать значения NEARSTACK (по умолчанию) или FARSTACK. В первом случае области данных и стека размещаются в одном и том же физическом сегменте, во втором — в разных.

Директива .MODEL делает доступными несколько идентификаторов, которые программист может задействовать в своей программе:

@code — физический адрес сегмента кода;

@data — физический адрес сегмента данных типа near (ближний);

@fardata — физический адрес сегмента данных типа far (дальний);

@fardata? — физический адрес сегмента неинициализированных данных типа far;

@curseg — физический адрес сегмента неинициализированных данных типа far;

@stack — физический адрес сегмента стека.

2.5. Разработка нашей первой программы на ассемблере

Вплоть до дня 7 мы будем с вами писать программы, предназначенные для выполнения в операционной системе MS-DOS, потому что так проще изучить ассемблер. Программы для MS-DOS не могут выполняться в операционной системе Windows, потому что используют 16-разрядную модель памяти реального режима (посмотрите еще раз описание памяти в разд. 1.6), поэтому такие программы еще называют 16-разрядными. Но как уже говорилось, Windows выполняет 16-разрядные программы во встроенной виртуальной машине VDM. На седьмом дне мы научимся писать 32-разрядные программы (использующие 32-разрядную модель памяти защищенного режима), предназначенные непосредственно для выполнения в ОС Windows.

16-разрядные программы MS-DOS с точки зрения программиста отличаются от 32-разрядных Windows-программ не только типом используемой памяти, но внутренней структурой, они даже транслируются и компонуются по-разному.

В MS-DOS существует два основных формата исполняемых файлов – COM и EXE. Об их отличиях мы поговорим далее подробно.

Windows также использует формат EXE для исполняемых файлов, но он имеет более сложную структуру в отличие от 16-разрядных программ.

Везде где мы будем далее говорить о EXE файлах вплоть до дня 7, то будем иметь только EXE формат файлов системы MS-DOS.

По существующей традиции самая первая программа на любом языке программирования должна выводить на экран приветствие "Hello, World!". Не будем отступать от этой традиции.

2.5.1. Программа типа COM

Наберите в любом текстовом редакторе текст, показанный в листинге 2.1, и сохраните его под именем hello1.asm.

Листинг 2.1. Программа типа COM (hello1.asm)

```
        .model   tiny
        .code
        org      100h
start:  mov      ah,9
        mov      dx,offset message
        int      21h
        ret
message          db  "Hello, World!",0Dh,0Ah,'$'

        end      start
```

Для получения исполняемого файла необходимо выполнить трансляцию и линковку.

В пакет MASM входит программа ml, которая позволяет выполнить автоматически трансляцию и линковку, для этого в командной строке необходимо выполнить такую команду:

```
ml hello1.asm
```

Если трансляция и линковка пройдет без ошибок, то в текущем каталоге появится файл hello1.com размером 24 байта. Если его выполнить, на экране появится фраза "Hello, World!" и программа сразу завершится.

Рассмотрим исходный текст программы, чтобы понять, как она работает.

Директива `.model tiny` устанавливает модель памяти типа TINY.

Для COM-файлов отведена *единственная* модель памяти типа TINY, все остальные модели памяти используются *только* в файлах типа EXE.

COM-файлы не содержат в себе никакой служебной информации, только код программы. Поэтому в отличие от EXE-файлов они хранятся на диске точно в том же виде, что и в памяти. Кроме того, COM-файл ни при каких обстоятельствах не может превышать размер 64 Кбайт (точнее немного меньше), так как он загружается в единственный сегмент, который одновременно является сегментом кода, данных и стека (модель памяти TINY). Директива `.code` начинает сегмент кода, который одновременно будет и сегментом данных и сегментом стека.

При загрузке в память COM-файл должен освободить первые 256 байт (100h) в памяти, куда MS-DOS поместит специальную служебную информацию, так называемый *префикс программного сегмента (PSP)*. Поэтому любая программа, предназначенная для ассемблирования в COM-файл должна начинаться директивой `org 100h`.

Метка `start` располагается перед самой первой командой в программе.

Команда `mov ah,9` заносит в регистр AH число 9 — это номер функции DOS вывода строки на экран (адрес строки должен быть задан в регистрах DS:DX). Команда `mov dx,offset message` помещает в регистр DX адрес строки `message`. Сегментный регистр DS будет автоматически заполнен при запуске COM-файла.

Команда `int 21h` далее вызывает эту функцию на выполнение.

В MS-DOS строки выводится на экран до первого встреченного символа $, поэтому строка "Hello, World!" завершается этим символом. Как поступить, когда нужно вывести на экран сам символ $ мы узнаем позже.

Код 0Dh является управляющим символом ASCII "возврат каретки", а код 0Ah символом ASCII "перевод строки". Эти два управляющих символа переводят курсор на первую позицию следующей строки.

Команда RET предназначена для выхода из процедуры, но в COM-файлах эта команда корректно завершает программу.

Директива END должна завершать код программы. Вместе с этой директивой всегда указывается метка, с которой начинается выполнение программы (`start` в нашем случае).

Отмечу, что большинство примеров программ реального режима далее в книге будут рассчитаны на компиляцию в COM-файлы, но вы легко сможете переписать их в EXE-формат, если такая необходимость появится.

После загрузки программы типа COM ее образ в памяти будет выглядеть, как показано на рис. 2.1.

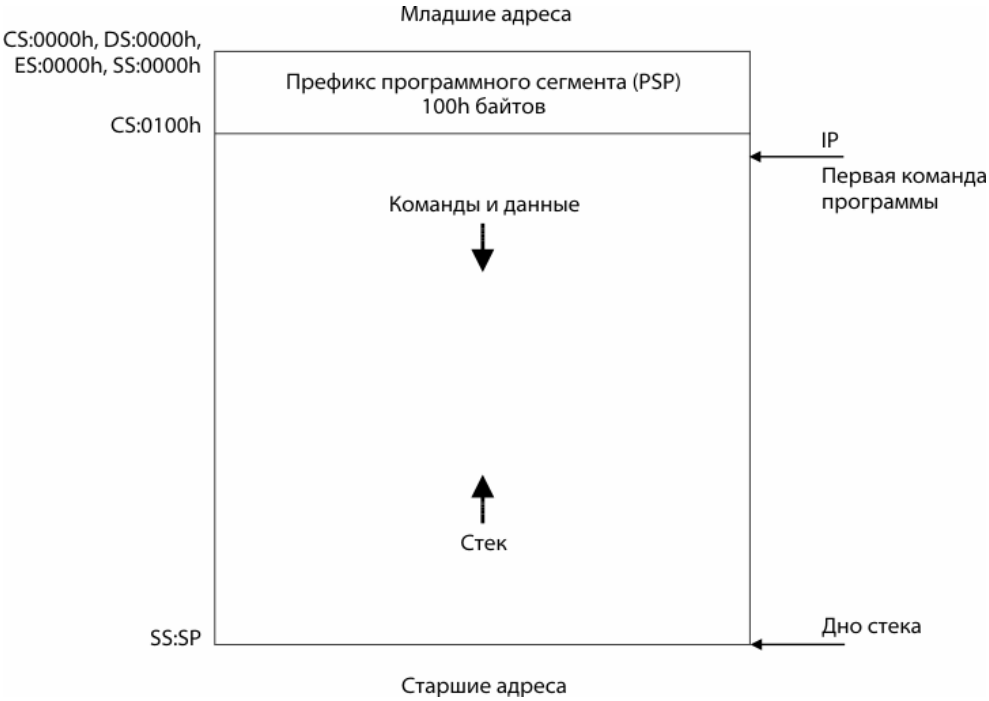

Рис. 2.1. Образ памяти программы типа COM

Образ программы начинается с префикса программного сегмента (PSP), который создается и заполняется системой. Объем PSP всегда равен 256 байтам. PSP содержит данные, используемые системой при выполнении программы (в разд. 2.8 структура PSP будет рассмотрена подробно). Как видите, программа типа COM состоит из единственного сегмента, в котором вместе располагаются код, данные и стек. Сегментные регистры (CS, DS, ES, SS) инициализируются автоматически и содержат одно и то же значение, указывающее на начало PSP. Значение регистра IP=100h.

2.5.2. Программа типа EXE

В листинге показана EXE-программа, которая выводит на экран приветствие: "Hello, World!". Сравните ее с COM-программой, как видите, программа типа EXE имеет немного более сложный вид, зато для EXE-файлов отсутствует ограничение в 64 Кбайт, поэтому все большие программы используют именно этот формат.

Листинг 2.2. Программа типа EXE (hello2.asm)

```
        .model small
        .stack 100h
        .code
start:  mov   ax,@data
        mov   ds,ax
        mov   ah,9
        mov   dx,offset message
        int   21h
        mov   ax,4C00h
        int   21h

        .data
message       db     "Hello, World!",0Dh,0Ah,'$'

        end   start
```

Компиляция и линковка осуществляется точно так же, как и в предыдущем примере, командой:

```
ml hello2.asm
```

Программа ml автоматически определит, какой файл требуется создать — типа EXE или типа COM.

В результате будет получен файл hello2.exe размером 546 байт. Как видите размер EXE-файла почти в 23 раза больше файла типа COM, хотя оба они выполняют одну и ту же функцию — выводят на экран фразу "Hello, World!". Это плата за размещение служебной информации (заголовка) внутри EXE-файла. Заголовок в EXE файлах составляет 512 байт, несложно подсчитать, что без учета заголовка код занимает всего 34 байта (546-512=34).

Рассмотрим исходный текст EXE-программы, чтобы понять, как она работает.

Директива `.model small` устанавливает модель памяти типа SMALL, а это значит, что мы можем разбить нашу программу на три сегмента: сегмент стека устанавливается директивой .STACK, сегмент кода начинается с директивы .CODE, а сегмент данных с директивы .DATA (в листинге я выделил эти директивы полужирным шрифтом).

Директива .STACK сразу позволяет задать размер стека, который рекомендуется устанавливать не короче 100h. Даже если вы в своей программе не будете использовать стек, его все равно необходимо задать, т. к. стек будет использовать MS-DOS.

В каждой EXE-программе сразу после метки начала кода (в нашем случае `start`), должны стоять следующие две команды:

```
mov ax,@data
mov ds,ax
```

Они необходимы, чтобы загрузить в сегментный регистр DS адрес сегмента данных (.data), иначе мы не сможем обращаться в программе к данным (у нас в сегменте данных только одна строка `message`). Для сегментных регистров CS и SS подобные операции делать не нужно, так как MS-DOS заносит адреса сегментов кода и стека в регистры CS и SS автоматически.

Предопределенная метка @data по сути является адресом сегмента данных. Вас наверное смущает почему мы не написали сразу более логичную на первый взгляд операцию:

```
mov ds,@data
```

Однако это нельзя сделать, т. к. в сегментные регистры можно загружать данные только из какого-либо другого регистра. Поэтому мы сначала загружаем адрес в регистр AX, а уже из регистра AX копируем в DS.

Программы типа EXE должны завершаться особым образом — вызовом DOS-функции 4Ch:

```
mov ax,4C00h
int 21h
```

Здесь мы одной командой `mov ax,4C00h` помещаем в регистр AH значение 4Ch, а в регистр AL — код возврата (значение 0), а команда `int 21h` вызывает функцию на выполнение.

После загрузки программы типа EXE ее образ в памяти будет выглядеть так, как показано на рис. 2.2.

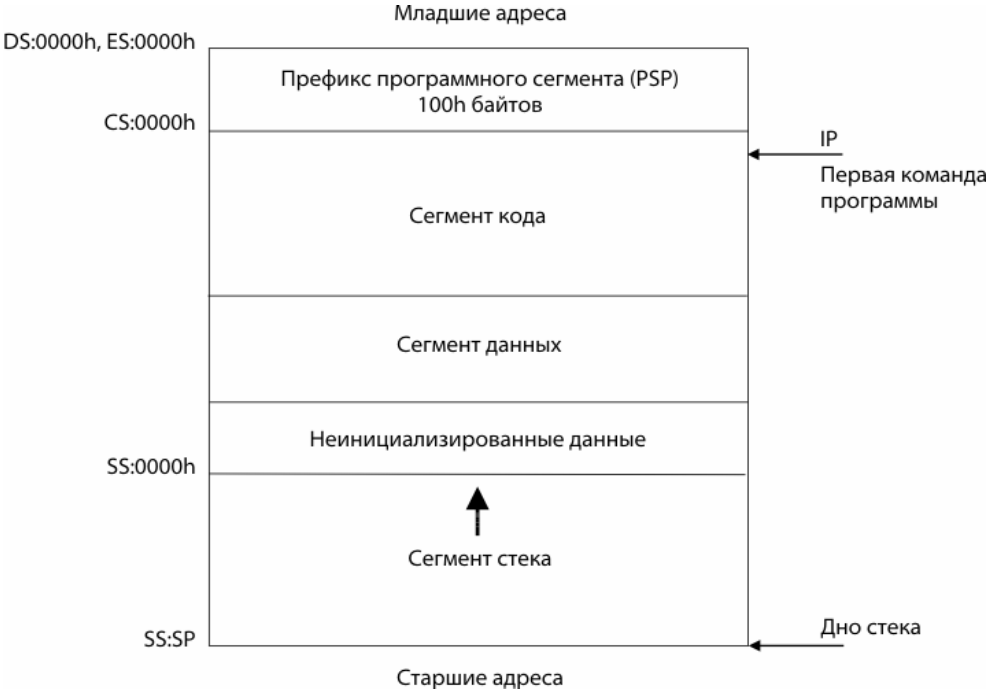

Рис. 2.2. Образ памяти программы типа EXE

Как и в COM-файлах образ EXE-программы начинается с префикса программного сегмента (PSP), который создается и заполняется системой. Объем PSP всегда равен 256 байтам. PSP содержит данные, используемые системой при выполнении программы (в разд. 2.8 структура PSP будет рассмотрена подробно). Сегментные регистры автоматически инициализируются значениями следующим образом:

- CS устанавливается на начало сегмента кода, в IP при этом загружается относительный адрес первой команды;
- DS и ES указывают на начало PSP, что позволяет программе с помощью этих значений обращаться к содержимому PSP;
- SS указывает на начало сегмента стека, в SP при этом загружается смещение конца сегмента стека.

2.6. Основные различия между программами типа EXE и COM

В таблице 2.1 представлены основные отличия программ типа COM и EXE.

Таблица 2.1. *Основные различия между программами типа COM и EXE*

	Программа типа COM	**Программа типа EXE**
Максимальный размер файла (в байтах)	65536 минус 256 (префикс) минус 2 (стек), таким образом, максимальный размер составляет 65278 байт	Неограничен
Служебная информация в файле (заголовок)	Отсутствует	512 байт (располагается непосредственно перед кодом программы)
Максимальное количество сегментов	Только один	Несколько сегментов кода и несколько сегментов данных. Сегмент стека всегда только один
Модель памяти	Только TINY	Любая кроме TINY
Размер стека (в байтах)	Генерируется автоматически и составляет: 65536 минус 256 (префикс) минус размер выполняемой программы и данных	Определяется программистом в сегменте стека (директива .stack). Должен быть не меньше 100h
Содержимое стека при входе	Слово со значением 0	Инициализирован или неинициализирован
Точка входа	PSP:100h	Определяется оператором END
IP при входе	100h	Относительный адрес точки входа
Содержимое регистра CS при входе	Адрес PSP	Адрес сегмента, содержащего точку входа
Содержимое регистра DS при входе	Адрес PSP	Адрес PSP
Содержимое регистра ES при входе	Адрес PSP	Адрес PSP
Содержимое регистра SS при входе	Адрес PSP	Адрес сегмента стека
Содержимое регистра SP при входе	Адрес вершины доступной памяти	Размер стека
Способ завершения	`ret`	`mov ax,4C00h` `int 21h`

2.7. Функции BIOS и DOS

У программиста под MS-DOS существует три способа работы с устройствами:

1. Непосредственная работа на самом низком уровне с использованием портов ввода-вывода.
2. С помощью вызова функций BIOS.
3. С помощью вызова функций DOS.

Непосредственная работа с устройствами осуществляется быстрее, чем через функции BIOS и DOS. Но работа с функциями BIOS и DOS проще и надежней, поэтому программисты на ассемблере используют в основном их. Необходимость программировать устройства напрямую через порты ввода-вывода возникает в следующих случаях:

1. При переключении процессора в "защищенный" режим, так как функции DOS и BIOS в этом случае становятся недоступными.

2. Для ускорения операций ввода-вывода — это требуется в основном в измерительных системах и системах, работающих в режиме реального времени.
3. При программировании нестандартного, нового, либо устаревшего оборудования для которого нет соответствующих функций BIOS и DOS.
4. Если требуется повышенная защита информации, например, при использовании крипто-ключа подключаемого к LPT, COM, либо USB-порту компьютера.

В целом программировать устройства напрямую через порты ввода-вывода несложно, но нужно обладать техническим описанием на устройство, чтобы знать, что и в какой последовательности нужно послать в порт и считать из него, и как следует трактовать эту информацию.

Функции BIOS и DOS при обращении к устройствам тоже работают с портами ввода-вывода, но эта работа скрыта от программиста. Все, что нужно сделать программисту — это просто вызвать функцию и считать возвращаемый ею результат, а функция все сделает самостоятельно. Существуют тысячи различных функций BIOS и DOS (где взять их описание будет рассказано ниже).

Все функции BIOS и DOS вызываются с помощью программных прерываний (командой INT). Прерывания мы будем подробно рассматривать в разд. 6.5. Сейчас лишь стоит отметить, что в реальном режиме процессора существует всего 256 прерываний, они нумеруются от 0h до FFh (от 0 до 255 в десятичной нотации).

Функции MS-DOS используют всего 32 прерывания с номерами в диапазоне от 20h до 3Fh (с 32-го по 63-е в десятичной нотации).

Функций BIOS используют диапазон прерываний с 10h по 1Fh (с 16-го по 31-е в десятичной нотации) и с 40h по 4Ah (с 64-го по 74-е в десятичной нотации), а также прерывание под номером 5h.

Несмотря на то, что функции BIOS и DOS используют лишь небольшие диапазоны прерываний, с помощью этих прерываний можно вызвать тысячи различных функций.

Например, с помощью прерывания `INT 21h` можно вызывать несколько сотен различных функций MS-DOS. Номер нужной функции обычно помещается в регистр AH перед вызовом прерывания. Кроме этого большинство функций требуют чтобы в определенные регистры были помещены различные необходимые параметры (аргументы функции).

Например, вспомним, как мы вызывали функцию DOS 09h:

```
mov  ah,9                    ; номер функции 09h
mov  dx,offset message       ; аргумент функции
int  21h                     ; вызываем прерывание 21h
```

После того как функция выполнит свою работу, некоторые регистры будут содержать возвращаемые значения. Результат работы большинства функций возвращается в регистр AL. Например, в результате успешного выполнения функции DOS 09h в регистре AL окажется значение 24h (код символа $).

Таким образом, чтобы использовать какую-либо функцию в своей программе вы должны знать ее формат, т. е. в какой регистр какое значение нужно занести перед вызовом функции и из какого регистра можно считать результат выполнения функции.

Описание всех функций DOS и BIOS можно найти в каких-нибудь старых справочниках по программированию под MS-DOS, а также в интернете в знаменитом "листе прерываний Ральфа Брауна" (Ralf Brown's Interrupt List) по адресу **http://www.cs.cmu.edu/~ralf/files.html** или по адресу **http://www.ctyme.com/rbrown.htm**. Лист прерываний Ральфа Брауна содержит описания всех прерываний от 0 до 255 и соответственно всех функций DOS и BIOS, которые используют определенные прерывания, но эти описания на английском языке. Последняя версия листа прерываний Ральфа Брауна имеет номер 61 (Release 61) и, по-видимому, этот лист обновляться уже не будет никогда.

Я сначала хотел сделать в конце этой книги приложение с описаниями всех функций DOS и BIOS, но отказался от этой затеи, т. к. во-первых, по количеству страниц такое приложение получилось бы в несколько раз больше основной части книги, а во-вторых, программирование под MS-DOS уже не актуально. Напомню, мы изучаем основы программирования под MS-DOS, лишь только для того чтобы нам проще

было освоить ассемблер. Впрочем, в этой книге мы познакомимся со всеми основными функциями DOS и BIOS.

2.8. Префикс программного сегмента (PSP)

Префикс программного сегмента (PSP, Program Segment Prefix) в ОС MS-DOS предшествует в памяти каждой COM или EXE-программе, и занимает 256 (100h) байт, причем структура PSP одинакова для файлов обоих типов. В COM-программе для PSP необходимо освобождать память директивой org 100h, в программах типа EXE этого делать не требуется. Префикс программного сегмента строит MS-DOS перед загрузкой в память каждой программы и использует его в своих целях, а также предоставляет возможность использовать PSP загруженной программе. Некоторые поля PSP существуют только лишь для совместимости с ранними версиями DOS и с древней операционной системой CP/M.

В табл. 2.2 показана структура префикса программного сегмента.

Я не буду подробно описывать назначение каждого поля, т. к. вряд ли вам это понадобится при программировании в настоящее время, но иметь общее представление о структуре PSP полезно. Единственное вам может оказаться полезным поле, расположенное по смещению PSP:80h, в котором сохраняются параметры командной строки (в том случае если они были переданы программе при запуске). Подробнее о том, как считать параметры командной строки рассказано в разд. 6.1.

С PSP тесно связаны структуры FCB (File Control Block — блок управления файлом) и DTA (Disk Transfer Address — область передачи данных). Фактически эти структуры являются частью PSP.

Есть два вида FCB: обычный и расширенный, расположенные соответственно по смещениям PSP:005Ch и PSP:006Ch. В табл. 2.3 приводится структура обычного FCB (всего 37 байт). В табл. 2.4 показан расширенный FCB (всего 44 байта). Эти структуры существуют только для совместимости со старыми версиями MS-DOS и CP/M.

Структура DTA (общий размер 43 байта) располагается по смещению PSP:0080h. Однако адрес начала этой структуры может быть переопределен с помощью функции 1Ah (INT 21h). Структура DTA заполняется и используется функциями поиска файлов.

Таблица 2.2. *Структура префикса программного сегмента (PSP)*

Относительный адрес	**Размер поля (в байтах)**	**Описание поля PSP**
0000h	2	Команда INT 20h (0CD20h). Если COM-программа завершается командой RETN, управление передается на эту команду. Существует для совместимости с системой CP/M
0002h	2	Адрес следующего сегмента в памяти, расположенного после программы. Это поле можно использовать, чтобы узнать размер памяти доступной программе (вычитая адрес, на которое оно указывает от смещения 0000 PSP). Значение возвращается в параграфах, поэтому его следует умножить на 16, чтобы получить размер в байтах
0004h	1	Зарезервировано
0005h	5	Длинный вызов (CALL FAR) диспетчера функций DOS. Существует для совместимости с системой CP/M
000Ah	4	Предыдущее содержимое вектора прерывания INT 22h (выход из программы). Если программа изменит этот вектор для своих целей, то после ее завершения DOS восстановит исходный вектор из PSP
000Eh	4	Предыдущее содержимое вектора прерывания INT 23h (обработчик нажатия Ctrl-Break). Если программа изменит этот вектор для своих целей, то DOS восстановит исходный вектор из PSP

Таблица 2.2. *(окончание)*

0012h	4	Предыдущее содержимое вектора прерывания INT 24h (обработчик критических ошибок). Если программа изменит этот вектор для своих целей, то DOS восстановит исходный вектор из PSP
0016h	2	Сегментный адрес PSP программы, которая запустила данную программу (программы-родителя). Обычно программой-родителем является COMMAND.COM
0018h	20	Таблица открытых файлов (JFT, Job File Table). Это массив из 20 однобайтных элементов, где один байт отводит на идентификатор. В свободных элементах таблицы находятся байты 0FFh
002Ch	2	Сегментный адрес копии окружения программы, в котором содержится набор ASCIIZ-строк, задающих значения некоторых глобальных переменных
002Eh	4	Адрес стека SS:SP программы (при последнем вызове INT 21h)
0032h	2	Максимальное число открытых файлов (число элементов JFT) (по умолчанию 20)
0034h	4	Дальний адрес JFT (по умолчанию PSP:0018)
0038h	4	Дальний адрес предыдущего PSP
003Ch	1	Флаг, указывающий, что консоль находится в состоянии ввода 2-байтного символа
003Dh	1	Флаг, устанавливаемый функцией 0B711h прерывания 2Fh (при следующем вызове INT 21h для работы с файлом имя файла будет заменено новым)
003Eh	2	Зарезервировано
0040h	2	Версия DOS, которую вернет функция DOS 30h
0042h	12	Зарезервировано
0050h	2	Команда INT 21h (0CD21h)
0052h	1	Команда RETF (0CBh)
0053h	2	Зарезервировано
0055h	7	Область для расширения первого FCB
005Ch	16	Первый FCB (File Control Block), заполняемый из первого аргумента командной строки. Это устаревший способ для получения доступа к файлам (существует для совместимости со старыми версиями DOS и CP/M). Смотрите структуру FCB в табл. 2.3
006Ch	16	Второй FCB, заполняемый из второго аргумента командной строки
007Ch	4	Зарезервировано
0080h	128	Это поле может содержать либо командную строку (точнее параметры командной строки с пробелами), либо область DTA (disk transfer area). Эти данные не могут существовать вместе, поэтому если необходимы параметры командной строки, то их необходимо сохранить после старта программы в надежное место (в какую-нибудь переменную в программе). Первый байт командной строки, который расположен по адресу PSP:80h содержит длину командной строки, а дальше находятся реальные параметры. Структура DTA показана в табл. 2.5

Таблица 2.3. *Обычный блок управления файлом (FCB)*

Относительный адрес	**Размер поля (в байтах)**	**Описание поля FCB**
0000h	1	Буква привода
0001h	8	Имя файла с пробелами
0009h	3	Расширение файла с пробелами
000Ch	2	Номер текущего блока
000Eh	2	Логический размер записи
0010h	4	Размер файла
0014h	2	Дата создания/обновления файла
0016h	2	Время создания/обновления файла
0018h	8	Зарезервировано
0020h	1	Номер текущей записи
0021h	2	Номер относительно записи

Таблица 2.4. *Расширенный блок управления файлом (FCB)*

Относительный адрес	**Размер поля (в байтах)**	**Описание поля FCB**
0000h	1	0FFh (сигнатура расширенного FCB)
0001h	5	Зарезервировано (нули)
0006h	1	Атрибуты файла
0007h	1	Буква привода
0008h	8	Имя файла с пробелами
0010h	3	Расширение файла с пробелами
0013h	2	Номер текущего блока
0015h	2	Логический размер записи
0017h	4	Размер файла
001Bh	2	Дата создания/обновления файла
001Dh	2	Время создания/обновления файла
001Fh	8	Зарезервировано
0027h	1	Номер текущей записи
0028h	2	Номер относительно записи

Таблица 2.5. *Структура DTA*

Относительный адрес	**Размер поля (в байтах)**	**Описание поля DTA**
0000h	1	биты 0-6: ASCII-код буквы привода бит 7: диск сетевой
0001h	11	Шаблон (маска) поиска (без пути)
000Ch	1	Атрибуты для поиска
000Dh	2	Порядковый номер файла в директории
000Fh	2	Номер кластера начала внешней директории
0011h	4	Зарезервировано

Таблица 2.5. *(окончание)*

0015h	1	Атрибут найденного файла
0016h	2	Время создания файла в формате DOS: биты 15-11: час (0-23) биты 10-5: минута биты 4-0: номер секунды, деленный на 2 (0-30)
0018h	2	Дата создания файла в формате DOS: биты 15-9: год, начиная с 1980 биты 8-5: месяц биты 4-0: день
001Ah	4	Размер файла
001Eh	13	ASCIIZ-имя найденного файла с расширением

2.9. Знакомство с отладчиком

Как говорилось в разд. 2.2 последний этап разработки программы это ее отладка. Хотя этап отладки является необязательным, однако нам необходимо научиться пользоваться отладчиком, т. к. отладчик лучше нам позволит понять некоторые важные тонкости ассемблера. К тому же сложно представить программиста на ассемблере (и вообще любого программиста), который не умел бы пользоваться отладчиком.

Существует большое количество отладчиков под MS-DOS, наиболее известными из них являются: Turbo Debugger от фирмы Borland, CodeView от Microsoft, AFDPRO. Существует даже стандартный консольный отладчик, встроенный во всех версиях DOS/Windows, который вызывается командой debug.exe, но он очень убогий.

Чтобы вам нигде не искать и не скачивать отладчик, мы будем использовать только отладчик CodeView, т. к. он стандартно входит в пакет MASM и обычно расположен в поддиректории \BINR (файл cv.exe). У большинства отладчиков пользовательский интерфейс подобен CodeView, поэтому если вы научитесь пользоваться CodeView, то без проблем при желании сможете воспользоваться любым другим отладчиком.

Замечу, что отладчики под DOS не подходят для отладки Windows-приложений. Под Windows можно посоветовать такие популярные отладчики как SoftIce и OllyDbg.

В качестве примера загрузим файл `hello1.com` (листинг 2.1) в отладчик CodeView. Для этого скопируйте файл в каталог отладчика (\BINR) и запустите из командной строки следующим образом:

```
cv hello1.com
```

Отладчик CodeView у вас должен выглядеть примерно так, как на рис. 2.3.

Сразу хочу предупредить, что пока загружен DOS-отладчик Windows будет скорее всего заметно подтормаживать, увы, с этим нельзя ничего поделать.

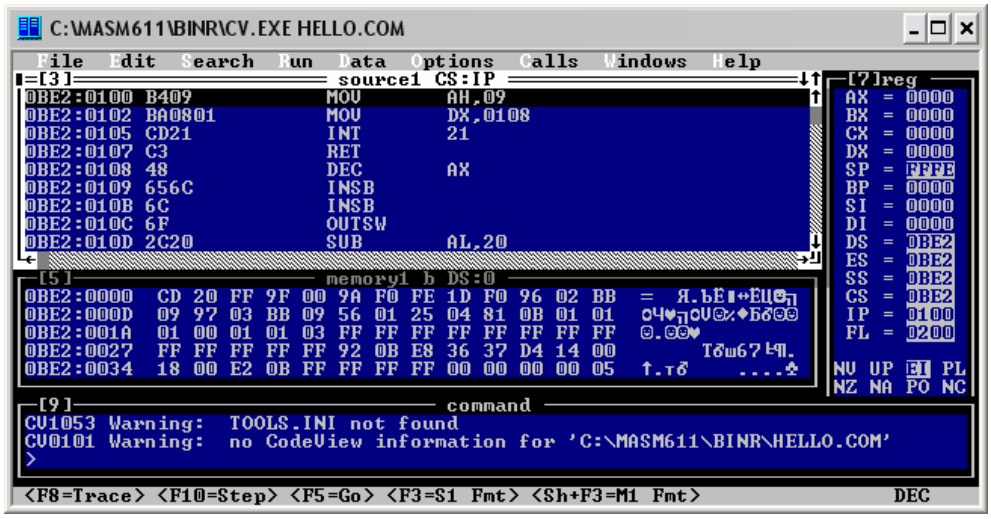

Рис. 2.3. Программа, запущенная в отладчике CodeView

Как видно экран отладчика состоит из нескольких окон. У каждого окна в левом верхнем углу стоит свой уникальный номер. Обычно по умолчанию открыты следующие окна:

- Окно 3 — дизассемблированный код.
- Окно 5 — содержимое памяти (в шестнадцатеричных кодах и в соответствующих им ASCII-символах).
- Окно 7 — регистры процессора.
- Окно 9 — командная строка.

С помощью меню отладчика "Windows" (меню расположено в самом верху отладчика) можно открыть другие окна отладчика, просто выбирая пункты меню с соответствующими названиями. Пункты меню можно выбирать мышью или нажатием клавиши <Alt> и выделяя стрелками нужный пункт с последующим нажатием <Enter> для выбора.

Закрыть любое окно можно, щелкнув мышью в его левом верхнем углу.

Посмотрим содержимое окна регистров (окно 7). Как видите сразу после загрузки регистр IP=100h, а сегментные регистры CS, DS, ES, SS имею одно и то же значение равное адресу PSP (см. табл. 2.1). В нижней части окна регистров показаны значения флагов процессора. Отладчик CodeView использует особые условные обозначения, показывающие состояния флагов (табл. 4.1).

Обычно по умолчанию в окне регистров показываются только 16-битные регистры (AX, BX, CX, ...), но вы можете включить в меню "Options" пункт "32-bit registers" для отображения 32-битных регистров (EAX, EBX, ECX, ...).

Таблица 4.1. *Перечень и значения флагов, выводимых в отладчике CodeView*

Название флага	**Выводимые значения CodeView**	
	Флаг установлен	**Флаг не установлен**
Левый столбец		
Флаг переполнения OF	OV	NV
Флаг прерывания IF	EI	DI
Флаг нуля ZF	ZR	NZ
Флаг паритета PF	PE	PO
Правый столбец		
Флаг направления DF	DN	UP
Флаг знака SF	NG	PL
Флаг вспомогательного переноса AF	AC	NA
Флаг переноса CF	CY	NC

Теперь посмотрим на содержимое окна 3 отладчика. Как видно оно состоит из четырех столбцов:

Адреса	Байты	Инструкции	Операнды
12BD:0100	B409	MOV	AH,09

Первый столбец содержит адреса в формате СЕГМЕНТ:СМЕЩЕНИЕ. Второй столбец машинные коды инструкций и операндов в шестнадцатеричном виде. Третий столбец имена инструкций. Четвертый столбец операнды инструкций.

Нажимая клавишу <F10>, вы можете выполнять трассировку программы, т. е. построчное выполнение программы, но без захода в процедуры. Для выполнения трассировки с заходом в процедуры нужно использовать клавишу <F8>. При этом можно наблюдать, как в окне регистров (окно 7) будут изменяться текущие значения регистров и флагов.

Можно выполнить сразу целый фрагмент программы (несколько строк). Для этого надо установить курсор в окне 3 перед той строкой, на которой требуется сделать остановку, и нажать клавишу <F7>. Выполнятся все строки программы до той, на которой установлен курсор; подсветка переместится на эту строку. Далее можно опять выполнять программу построчно, нажимая клавишу <F10> или, установив в требуемом месте курсор, выполнить следующий фрагмент, нажав <F7>.

В любое время можно вернуть программу в первоначальное состояние (какое было в момент загрузки), т. е. выполнить рестарт. Для этого надо в главном меню выбрать пункт **Run**, а в подменю этого пункта — пункт **Restart**.

Если программа должна выводить что-то на экран, то, не выходя из отладчика, можно увидеть результат работы программы. Для этого нужно выбрать меню **View** и в нем подменю **Output,** или просто нажать клавишу <F4>. Для возврата в окно отладчика можно нажать любую клавишу.

С помощью меню **File –> Exit** можно выйти из отладчика. Выход из отладчика можно осуществить также нажатием комбинации клавиш <Alt>+<F4>.

2.10. Младший байт по младшему адресу

Отладчик сейчас нам поможет понять важный принцип размещения данных в памяти, который вы должны хорошо знать.

Если запись в память осуществляется отдельных байтов, то они располагаются в естественном порядке — в порядке возрастания адресов памяти. Но если осуществляется запись многобайтовых единиц информации, то байты располагаются в памяти в обратном порядке по принципу: **младший байт по младшему адресу**. Эта особенность не языка ассемблера, а архитектуры микропроцессоров Intel.

Чтобы было более понятно, рассмотрим на конкретном примере (листинг 2.3).

Листинг 2.3. Программа для демонстрации размещения данных в памяти

```
        .model small
        .stack 100h

        .code
start:  mov    ax,@data
        mov    ds,ax
        mov    ah,9
        mov    dx,offset message1
        int    21h
        mov    ax,4C00h
        int    21h

        .data
message1        db      "Запустите программу в отладчике $"
```

```
var1          db      47h             ; однобайтовое значение
var2          dw      2a5ch           ; двухбайтовое значение
var3          dd      8f3d89a1h       ; четырехбайтовое значение
mas1          db      10 dup (" ")
mas2          db      5 dup (?)
message2      db      "Конец сегмента данных $"

        end     start
```

Запустите программу под отладчиком. Для этого скопируйте файл в каталог отладчика (\BINR) и запустите из командной строки следующим образом:

```
cv memdata.com
```

Затем поместите курсор в окне Dump и промотайте немного вниз, пока не увидите содержимое памяти сегмента данных нашей программы (рис. 2.4).

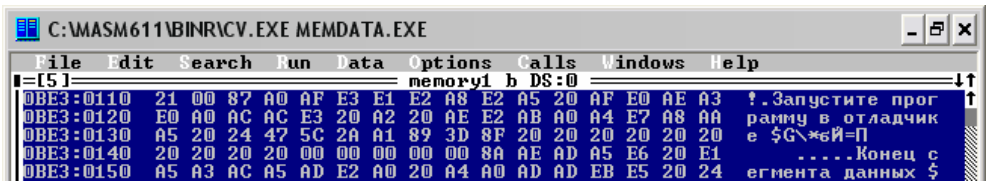

Рис. 2.4. Дамп памяти в окне отладчика CodeView

Как видно сразу за строкой "Запустите программу в отладчике $" в памяти располагается содержимое переменной var1 равное байту 47h. Далее обратите внимание, как размещены в памяти байты, входящие в слово переменной var2. Сначала следует байт 5ch, а затем байт 2ah. То есть значения расположены наоборот: сначала расположен младший байт значения, а затем старший. Точно также согласно принципу "младший байт по младшему адресу" расположены байты, входящие в двойное слово переменной var3. Затем идут десять пробелов массива mas1, затем пять случайных значений массива mas2 и затем строка message2.

Заметьте, что строки располагаются в памяти не в перевернутом виде, потому что определение вида:

```
message1      db      "Запустите программу в отладчике $"
```

Заставляет процессор рассматривать каждый символ как отдельный байт, поэтому строка не является многобайтным значением (посмотрите еще раз примеры в разд. 2.4.1).

Повторюсь, что согласно принципу "младший байт по младшему адресу" размещаются в памяти только *многобайтовые значения*, определяемые в основном с помощью таких директив как DW, DD, DF, DP, DQ и DT, но только не DB.

Когда вы будете выполнять операции с данными с помощью процессорных команд, таких как ADD — сложить, SUB — вычесть и пр., то вам не нужно задумываться о размещении данных в памяти, т. к. все преобразования будет осуществлять процессор.

Но если вы будете работать с оперативной памятью на физическом уровне, то необходимо учитывать возможное перевернутое расположение данных.

ДЕНЬ 3

Основные конструкции ассемблера

В этот день мы изучим все основные конструкции, с помощью которых строятся программы на ассемблере. К таким конструкциям относятся циклы, безусловные и условные переходы, подпрограммы, макросы и др.

3.1. Цикл

На втором дне мы с вами научились создавать программу, которая выводит на экран фразу "**Hello, World!**".

А теперь представим, что нам требуется создать программу, которая должна вывести на экран *пять раз* фразу "**Hello, World!**". Конечно, можно просто пять раз выполнить в программе операцию вывода, как показано в листинге 3.1.

Листинг 3.1. Вывод на экран пять раз "Hello, World!" без использования цикла (loop1.asm)

```
        .model tiny
        .code
        org     100h

start:  mov     ah,9
        mov     dx,offset message
        int     21h

        mov     ah,9
        mov     dx,offset message
        int     21h

        mov     ah,9
        mov     dx,offset message
        int     21h

        mov     ah,9
        mov     dx,offset message
        int     21h

        mov     ah,9
        mov     dx,offset message
        int     21h
        ret
message         db      "Hello, World!",0Dh,0Ah,'$'
        end start
```

Выполните трансляцию и линковку этой программы:

```
ml loop1.asm
```

Затем запустите полученный файл loop1.com, вы должны увидеть на экране:

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Программа успешно работает, но нерациональность такого решения очевидна. Нам приходится набирать один и тот же участок кода в пяти экземплярах. Возможно, пять раз не столь ощутимо, но представим, что нам бы потребовалось вывести 100 или даже 1000 раз фразу "**Hello, World!**"?

Чтобы решить эту проблему для выполнения повторяющихся действий в любом языке программирования присутствует оператор цикла. В ассемблере для создания циклов служит команда LOOP, которая имеет следующий синтаксис:

```
loop метка
```

Метка служит для передачи управления на ту команду (или последовательность команд), которая должна повторяться. Число повторов для команды LOOP всегда задается в регистре CX (для 16-битных программ) или в регистре ECX (для 32-битных программ) (неслучайно регистр CX называется "счетчик"). Обычно значение в CX заносится с помощью команды MOV CX,N, где N – целое число большее нуля.

В общем виде цикл на ассемблере с использованием LOOP организуется следующим образом:

```
MOV CX, N           ; устанавливаем число повторов (N>0)
Mark:               ; метка
       ...          ; тело цикла будет выполнять N раз
LOOP Mark
```

В листинге показана программа с применением цикла для вывода пять раз фразы "**Hello, World!**".

Листинг 3.2. Вывод на экран пять раз "Hello, World!" с использованием цикла (loop2.asm)

```
        .model tiny
        .code
        org    100h

start:
         mov    cx,5    ; устанавливаем число повторов (5 раз)
Mark:
        mov     ah,9                    ; тело цикла
        mov     dx,offset message       ;
        int     21h                     ;
        loop    Mark

        ret
message         db      "Hello, World!",0Dh,0Ah,'$'

        end     start
```

Как видите, программа получилась значительно короче, чем в листинге 3.1.

Файл после трансляции и линковки тоже получается меньше, чем в предыдущем примере, за счет того, что мы использовали меньшее количество инструкций.

Стоит отметить важную деталь, что после каждой итерации цикла значение в CX (или в ECX для 32-битных программ) уменьшается на 1. То есть, в нашем случае после вывода первой фразы "Hello, World!" в CX останется значение 4, после вывода второй фразы в CX останется 3 и т. д. Цикл завершается, когда в CX остается значение 0.

Настоятельно советую запустить эту программу под отладчиком и потрассировать программу клавишей <F8> (пошаговое выполнение с заходом в процедуры). При этом наблюдайте, как будет уменьшаться значение регистра CX в окне регистров отладчика (окно 7). Далее в книге я не буду напоминать, но старайтесь работу каждой программы из этой книги наблюдать в отладчике (см. разд. 2.9).

Существуют разновидности команды LOOP.

Следующие две команды работают одинаково (имеют одинаковые коды):

```
loope метка
loopz метка
```

Переход на метку осуществляется, если значение регистра CX ≠ 0 и значение флага ZF = 1.

Следующие две команды также имеют одинаковые коды, а, значит, работают одинаково:

```
loopne метка
loopnz метка
```

В них переход на метку осуществляется, если CX ≠ 0 и ZF = 0.

Также как для обычной команды LOOP число повторов для перечисленных четырех команд задается в регистре CX (для 16-битных программ) или в регистре ECX (для 32-битных программ). Содержимое регистра CX (ECX) уменьшается на один после каждой итерации, команды заканчивают выполнение, когда значение в CX (ECX) становится равным нулю.

3.2. Безусловный переход

В программах можно нарушать естественный (последовательный) порядок выполнения команд, для этого служат команды перехода. Переходы бывают условные и безусловные. Условный переход выполняется только тогда когда выполняется заранее заданное условие, безусловный переход выполняется всегда без каких-либо условий. В ассемблере безусловный переход выполняет команда:

```
JMP метка
```

Эта команда является аналогом оператора `goto` в языках высокого уровня. После выполнения JMP управление передается на метку (точнее на команду помеченную меткой).

Например, в листинге 3.3 показана программа, в которой присутствуют две функции для вывода на экран сначала фразы "Hello, World!", а затем "Goodbye, World!". Однако реально будет выведена на экран только вторая фраза, потому что с помощью команды JMP выполняет безусловный переход сразу на выполнение второй функции, т. е. первая функция вывода будет пропущена.

Листинг 3.3. Пример выполнения безусловного перехода (jump1.asm)

```
        .model tiny
        .code
        org    100h

start:
```

```
        jmp     Go_mess2

        mov     ah,9                    ; эта функция не будет выполнена
        mov     dx,offset message1      ;
        int     21h                     ;

Go_mess2:

        mov     ah,9
        mov     dx,offset message2
        int     21h

        ret

message1        db      "Hello, World!",0Dh,0Ah,'$'
message2        db      "Goodbye, World!",0Dh,0Ah,'$'

        end     start
```

В качестве домашнего задания: добавьте в листинг 3.3 дополнительные команды JMP, так чтобы программа сначала выводила на экран фразу "Goodbye, World!", а затем "Hello, World!". При этом удалять и переставлять существующие команды в программе не нужно. Сколько минимальное количество JMP для этого нужно добавить в программу?

В ассемблере символ $, используемый в качестве операнда в машинной команде, всегда соответствует текущему адресу.

Например, команда

```
jmp $
```

Создает вечный цикл, т. к. выполняет безусловный переход на саму себя.

Символ $ можно использовать для адресации. Например, в следующем примере команда JMP выполняет переход на команду MOV:

```
jmp $+4 ; безусловный переход на команду mov
nop ; длина команды NOP составляет 1 байт
nop ; длина команды NOP составляет 1 байт
mov bx,10
```

Это происходит потому, что сама команда JMP занимает 2 байта и две команды NOP в сумме дают 2 байта, в итоге требуется всего 4 байта, для того чтобы перепрыгнуть на команду MOV. Команда NOP (ее машинный код 90h) это так называемая ничего не делающая команда, она только занимает место и время.

В команде JMP может указываться тип перехода:

- short (короткий переход) — если адрес перехода находится в пределах -128…+127 байт от команды JMP. Команда короткого перехода занимает всего 2 байта: в первом байте записывается код операции (EBh), во втором – смещение к точке перехода.
- near (ближний переход) — если адрес перехода находится в пределах того же сегмента, что и команда JMP. В команде ближнего перехода под смещение отводится целое слово (это дает возможность осуществить переход в любую точку 64-кбайтного сегмента), поэтому она занимает 3 байта (в первом байте находится код операции – 0E9h). По умолчанию ассемблер транслирует команды JMP, именно в этой тип.
- far (дальний переход) — если адрес перехода находится в другом сегменте. Команда дальнего перехода имеет длину 5 байт. В первом байте находится код операции (0EAh), а в следующих четырех байтах — адрес сегмента и смещение точки перехода внутри сегмента.

Типы far и near в программе должны обязательно указываться с описателем PTR. Примеры:

```
jmp Lab              ; ближний переход (near) (3 байта)
jmp short Lab        ; короткий переход (2 байта)
jmp near ptr Lab     ; ближний переход (3 байта)
jmp far ptr Lab      ; дальний переход (5 байт)
jmp short $+2        ; это команда просто передает
                     ; управление на следующую команду.
```

3.3. Сравнение и условные переходы

Если переход осуществляется только при выполнении соответствующего условия, то такой переход называется условным. Команд условного перехода достаточно много в ассемблере, но все они имеют следующий общий вид:

```
Jcc метка
```

где операнд указывает метку той команды в программе, на которую нужно сделать переход в случае выполнения некоторого условия, а `cc` сокращение из одной или двух букв описывающее условие:

e — равно (equal),

n — не (not),

g — больше (greater),

l — меньше (less),

a — выше (above),

b — ниже (below).

Условием для команд условного перехода служат отдельные флаги в регистре флагов EFLAGS/FLAGS. Например, команда JZ проверяет флаг нуля, поэтому если в какой-либо программе встретится команда JZ и при этом в регистре флагов будет установлен в 1 флаг нуля (ZF=1), то произойдет переход на указанную метку, если же флаг нуля будет установлен в нуль (ZF=0), то переход не осуществится. В табл. 3.1 перечислены все команды условных переходов и флаги, значения которых они проверяют.

Таблица 3.1. *Команды условных переходов*

Команда	Условие для CMP	Значения флагов
ja jnbe	Если выше Если не ниже и не равно	CF=0 и ZF=0
jae jnb jnc	Если выше или равно Если не ниже Если перенос	CF=0
jb jnae jc	Если ниже Если не выше и не равно Если перенос	CF=1
jbe jna	Если ниже или равно Если не выше	CF=1 и ZF=1
jcxz jecxz	Если CX=0 Если ECX=0	Не влияет на флаги
je jz	Если равно Если ноль	ZF=1

Таблица 3.1. *(окончание)*

jg jnle	Если больше Если не меньше и не равно	ZF=0 и SF=OF
jge jnl	Если больше или равно Если не меньше	SF=OF
jl jnge	Если меньше Если не больше и не равно	SF<>OF
jle jng	Если меньше или равно Если не больше	ZF=1 или SF<>OF
jne jnz	Если не равно Если не ноль	ZF=0
jno	Если нет переполнения	OF=0
jnp jpo	Если нет четности Если нечетное	PF=0
jns	Если нет знака	SF=0
jp jpe	Если есть четность Если четное	PF=1
jo	Если есть переполнение	OF=1
js	Если есть знак	SF=1

Слова "выше" и "ниже" в таблице употребляются при сравнении беззнаковых целых чисел, слова "больше" и "меньше" учитывают знак. Например: число -1 меньше нуля, но если его рассматривать как беззнаковое, то оно будет выше нуля, потому что примет значение 65535 (0FFFFh). При выборе команды условного перехода в своей программе вы должны учитывать, будет выполняться сравнение знаковых или беззнаковых чисел.

Если команды условного перехода только проверяют флаги, то понятно, что должны присутствовать в ассемблере команды меняющие флаги. Обычно флаги меняют арифметические команды, устанавливая, например, флаг переноса или переполнения в результате своей работы (с арифметическими командами мы познакомимся на пятом дне). Эти флаги могут проверять соответствующие команды условного перехода, выполняя переход на метку или не выполняя. Однако чаще всего команды условного перехода работают совместно с командой сравнения (CMP происходит от англ. compare — сравнение):

```
CMP op1,op2
```

Эта команда сравнивает величины op1 и op2 и устанавливает соответствующие флаги в регистре флагов. На самом деле эта команда просто вычисляет разность op1-op2 без сохранения результата. Понятно, что возможны следующие варианты: op1<op2, op1<=op2, op1=op2, op1>=op2 и op1>op2. Таблица 3.1 поможет вам определить условия CMP, при которых срабатывают соответствующие переходы. Например, если операнды op1 и op2 не равны между собой, то команда JNE выполнит переход на указанную метку.

В листинге 3.4 показана программа, которая сравнивает два числа, задаваемые в регистрах AX и BX, и выводит результат сравнения на экран. Ассемблирование выполняется как обычно:

```
ml file.asm
```

После выполнения программа выдаст на экран результат:

```
Число в AX больше, чем в BX.
```

Сравнение осуществляется с учетом знака, поэтому можно в AX и BX задавать как положительные, так и отрицательные числа. Попробуйте подставлять различные значения и после ассемблирования посмотреть результаты.

Листинг 3.4. Сравнение двух чисел (jump2.asm)

```
        .model tiny
        .code
        org     100h

start:
        mov     ax,3
        mov     bx,-15

        cmp     ax,bx
        jle     Lab

        mov     ah,9
        mov     dx,offset message1
        int     21h
        ret
Lab:
        mov     ah,9
        mov     dx,offset message2
        int     21h

        ret

message1        db "Число в AX больше, чем в BX.",0Dh,0Ah,'$'
message2        db "Число в AX меньше или равно значению в BX.",0Dh,0Ah,'$'

        end     start
```

Команды CMP и Jcc вместе аналогичны оператору `if... then...`, который имеется практически во всех языках высокого уровня. Например, команда на языке BASIC:

```
if AX=>BX then goto Lab
```

на языке ассемблера будет выглядеть следующим образом:

```
cmp ax,bx
jge Lab
```

3.4. Стек

Основными двумя командами для работы со стеком являются: PUSH (от англ. — вталкивать) и POP (от англ. — выталкивать), имеющие следующий синтаксис:

```
PUSH источник
POP приемник
```

Команда PUSH записывает в стек свой операнд, а команда POP считывает слово с вершины стека и присваивает его указанному операнду.

Например, после выполнения следующих команд

```
mov ax,777      ; записать в AX значение 777
push ax         ; записать содержимое AX в стек
pop bx          ; извлечь из стека
```

в регистре BX окажется значение 777.

Кроме основных двух команд существуют команды для записи в стек и чтения из стека регистра флагов:

Для 16-битного режима:

PUSHF — записывает регистр флагов FLAGS в стек.

POPF — считывает регистр флагов FLAGS из стека.

Для 32-битного режима:

PUSHFD — записывает регистр флагов EFLAGS в стек.

POPFD — считывает регистр флагов EFLAGS из стека.

Эти команды используются без всяких операндов и обычно нужны для сохранения текущих состояний флагов (FLAGS или EFLAGS) и последующего их восстановления.

Понятно, что команды PUSHF и PUSHFD не меняют флаги, а команды POPF и POPFD, естественно, меняют все флаги.

Начиная с процессора 80186, появились команды для работы со стеком, которые позволяют поместить в стек сразу все регистры общего назначения:

Для 16-битного режима:

PUSHA — размещает в стеке регистры в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI (для SP берется значение, которое находилось в этом регистре до начала работы команды).

POPA — выполняет действие обратное PUSHA, но помещенное в стек значение SP игнорируется.

Для 32-битного режима:

PUSHAD — размещает в стеке регистры в следующем порядке: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI (для ESP берется значение, которое находилось в регистре до начала работы команды).

POPAD — выполняет действие обратное PUSHAD, но помещенное в стек значение ESP игнорируется.

Кроме того, процессоры 8086 и 8088 не позволяли в команде PUSH использовать непосредственный операнд, такая возможность впервые появилась в процессоре 80186.

Например:

```
push 7 ; такая команда допустима только в процессорах 80186 и выше
```

В процессорах 8086 и 8088 нужно использовать регистр:

```
mov ax,7
push ax
```

Так как по умолчанию ассемблер разрешает использовать только команды процессора 8086, то для того, чтобы можно было использовать команды процессора 80186, надо перед первой из таких команд в программе задать директиву .186 или выше (.286, .386, .586…):

```
.186
pusha           ; без директивы .186 ассемблер будет считать
                ; эту команду ошибочной
push 100h       ; без директивы .186 ассемблер будет считать
                ; эту команду ошибочной
pushf           ; эту команду можно использовать без директивы .186
```

3.5. Подпрограммы (процедуры)

В больших программах, чтобы избежать повторного написания группы из одних и тех же команд, их обычно оформляют в виде одной подпрограммы и вызывают по мере необходимости из любых мест программы.

Например, если часто нужно выводить на экран различные строки, то команды

```
mov  ah,9
int  21h
```

можно оформить в отдельную процедуру и вызывать ее тогда, когда требуется вывод на экран.

Подпрограммы оформляются с помощью двух директив: PROC (от англ. "procedure" — процедура) и ENDP (от англ. "end of procedure" — конец процедуры) следующим образом:

```
имя_процедуры PROC параметр
тело_процедуры
RET
имя_процедуры ENDP
```

В обеих директивах указывается одно и то же имя процедуры.

Директива PROC может иметь параметр — NEAR (от англ. близкий) или FAR (от англ. дальний). Если параметр отсутствует, то ассемблер будет считать, что он равен NEAR (в связи с этим параметр NEAR обычно не указывается). Соответственно процедура, оформленная с параметром NEAR (или с отсутствующим параметром) называется "близкой", а с параметром FAR — "дальней". К близкой процедуре нельзя обращаться из других сегментов команд, а к дальней — можно.

Вызов процедуры осуществляется с помощью команды CALL, которая имеет следующий синтаксис:

```
CALL имя_процедуры
```

А возврат из процедуры осуществляется с помощью команды RET.

В листинге 3.5 показана программа с применением процедуры. В процедуре (я дал ей имя Output) размещаются команды вызова функции DOS 9h. Процедура вызывается три раза для вывода на экран трех разных строк.

Листинг 3.5. Пример работы процедуры (proc.asm)

```
        .model tiny
        .code
        org    100h

start:
        mov    dx,offset message1
        call   Output

        mov    dx,offset message2
        call   Output

        mov    dx,offset message3
        call   Output

        ret
```

```
Output PROC
       mov    ah,9
       int    21h
       ret
Output ENDP

message1      db     "Hello, Ivan!",0Dh,0Ah,'$'
message2      db     "Hello, Petr!",0Dh,0Ah,'$'
message3      db     "Hello, Fedor!",0Dh,0Ah,'$'

       end    start
```

3.6. Директива INCLUDE

Синтаксис директивы:

```
INCLUDE имя_файла
```

Директива INCLUDE вставляет в текущий исходный код программы исходный код из указанного файла. В языке программирования Си есть аналогичная директива #include.

Заданное `имя_файла` должно указывать на существующий файл. Расширение включаемого файла не имеет значения, но содержимое включаемого файла должно иметь код на ассемблере. Ассемблер ищет файл в текущей директории. Если файл находится вне текущей директории, то необходимо указывать полный или частичный путь к файлу. Примеры:

```
INCLUDE file.asm                    ; Файл в текущей директории
INCLUDE local\as\entry.lib          ; Частичное наименование пути
INCLUDE d:\include\define.inc       ; Полный путь к файлу
```

Существует подобная директива:

```
INCLUDELIB имя_файла
```

Она указывает компоновщику имя дополнительной библиотеки или объектного файла, который потребуется при составлении данной программы. Мы эту директиву, равно как и директиву INCLUDE, будем часто использовать при изучении программирования под Windows.

Рассмотрим пример работы директивы INCLUDE. Создадим два файла: первый const.inc будет содержать несколько констант, а второй loop3.asm почти такой же, как в листинге 3.2, будет подключать с помощью INCLUDE файл const.inc и использовать из него константы (листинги 3.6 и 3.7).

Ассемблирование осуществляется как обычно выполнением командной строки:

```
ml loop3.asm
```

при этом оба файла const.inc и loop3.asm должны находиться в одной директории.

Листинг 3.6. Содержимое включаемого файла const.inc

```
; содержимое файла const.inc
       cr     EQU   0Dh
       lf     EQU   0Ah
       COUNT  EQU    10
```

Листинг 3.7. Содержимое файла loop3.asm

```
       .model tiny
```

```
        INCLUDE const.inc     ; включаем содержимое файла const.inc

        .code
        org     100h
start:
         mov    cx,COUNT       ; константа из файла const.inc
Mark:
        mov     ah,9
        mov     dx,offset message
        int     21h
        loop    Mark

        ret

; используются константы cr и lf из файла const.inc
message         db      "Hello, World!",cr,lf,'$'

        end start
```

3.7. Конструкции времени исполнения программы

Выше мы рассмотрели, как организуются циклы и условные переходы с помощью команд ассемблера. Но в ассемблере еще предусмотрены специальные директивы, которые могут значительно упростить программирование на ассемблере, т. к. позволяют писать циклы и условные переходы в наглядном виде, почти также как на языках высокого уровня. Конструкции созданные с помощью этих директив преобразуются на этапе ассемблирования в команды микропроцессора.

Обратите внимание, что перед каждой такой директивой обязательно ставится точка (аналогичные директивы без точки выполняют другие действия, о чем мы узнаем ниже при рассмотрении директив условного ассемблирования).

Мы конструкции времени исполнения будем активно использовать в программах под ОС Windows (день 7).

- Условные конструкции:

```
.IF условие
...
.ENDIF
```

Если условие "истина", то выполняются команды между .IF и .ENDIF.

Директива .IF может также содержать .ELSE:

```
.IF условие
...
.ELSE
...
.ENDIF
```

В этом случае если условие "истина", то выполняются команды между .IF и .ELSE, в противном случае — от .ELSE до .ENDIF.

Директива .IF может также включать множество операторов .ELSEIF:

```
.IF условие1
...
.ELSEIF условие2
...
.ELSEIF условие3
...
.ELSE
...
.ENDIF
```

Если условие "истина", то выполняются команды соответствующего участка конструкции. Например, если условие2 "истина", то будет ассемблироваться участок между первой и второй директивой .ELSEIF. Если все три условия "ложь", то выполняются команды от .ELSE до .ENDIF.

Пример условной конструкции:

```
.IF AX == 7
       mov bx,66h
.ELSE
       mov bx,33h
.ENDIF
```

Представленный фрагмент эквивалентен следующему ассемблерному коду:

```
cmp ax,7
jne NO
mov bx,66h
jmp EXIT
NO:
mov bx,33h
EXIT:
```

- Конструкция цикла:

```
.WHILE условие
...
.ENDW
```

Пример:

```
.WHILE AX<99h
add ax,10h
.ENDW
```

Представленный фрагмент эквивалентен следующему ассемблерному коду:

```
Label1:
       add ax,10h
Label2:
       cmp ax,99h
       jb Label1
```

В условиях можно использовать операторы из табл. 3.2.

Например, можно записать такое условие:

```
.IF (eax==1 && ebx!=2)
       ; если eax = 1 и ebx не равно 2
.ENDIF
```

Таблица 3.2. *Операторы условий в конструкциях времени исполнения программы*

Оператор	Условие
==	равно
!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно
&	проверка бита
!	инверсия (NOT)
&&	логическое 'И' (AND)
||	логическое 'ИЛИ' (OR)
CARRY?	флаг переноса (cf) установлен
OVERFLOW?	флаг переполнения (of) установлен
PARITY	флаг паритета (pf) установлен
SIGN?	флаг знака (sf) установлен
ZERO?	флаг нуля (zf) установлен

3.8. Директивы условного ассемблирования

Директивы условного ассемблирования проверяют условия только во время *ассемблирования* программы, т. е. служат только для указания определенных действий транслятору. Как мы уже знаем, в исполняемый файл директивы не включаются, поэтому никакие директивы в принципе не способны проверять рабочие условия во время выполнения программы.

Директива условного ассемблирования имеет следующую основную конструкцию:

```
IF выражение
...
ENDIF
```

Если значение выражения "истина" (не равно нулю), то ассемблируется участок между IF и ENDIF.

Директива IF может также содержать ELSE:

```
IF выражение
...
ELSE
...
ENDIF
```

В этом случае если выражение "истина" (не равно нулю), то ассемблируется участок между IF и ELSE, в противном случае — от ELSE до ENDIF.

Директива IF может также включать множество операторов ELSEIF:

```
IF выражение1
...
ELSEIF выражение2
...
```

```
ELSEIF выражение3
...
ELSE
...
ENDIF
```

Если выражение "истина" (не равно нулю), то ассемблируется соответствующий участок программы. Например, если выражение2 не равно нулю, то будет ассемблироваться участок между первой и второй директивой ELSEIF. Если все три выражения "ложь" (равны нулю), то ассемблируется фрагмент ELSE до ENDIF.

Существуют директивы, проверяющие специальные условия.

Директива IF1 обеспечивает ассемблирование только на первом проходе ассемблирования:

```
IF1 выражение1
...
ELSEIF1 выражение2
...
ELSE
...
ENDIF
```

Директива IF2 обеспечивает ассемблирование только на втором проходе ассемблирования:

```
IF2 выражение1
...
ELSEIF2 выражение2
...
ELSE
...
ENDIF
```

Директива IFE обеспечивает ассемблирование, если значение выражения оценивается как "ложное" (нуль):

```
IFE выражение1
...
ELSEIFE выражение2
...
ELSE
...
ENDIF
```

Директива IFDEF проверяет, было ли определено указанное 'имя' и обеспечивает ассемблирование только в том случае, если это 'имя' представляет собой некоторую метку, переменную или символьное имя:

```
IFDEF имя1
...
ELSEIFDEF имя2
...
```

```
ELSE
...
ENDIF
```

Директива IFNDEF обеспечивает ассемблирование, если указанное имя еще не было определено:

```
IFNDEF имя1
...
ELSEIFNDEF имя2
...
ELSE
...
ENDIF
```

Директива IFB проверяет значение указанного 'аргумента' и обеспечивает ассемблирование, если 'аргумент' пустой (пробел). Аргументом этой директивы может быть любое имя, число или выражение. Угловые скобки (<>) должны указываться обязательно:

```
IFB <аргумент1>
...
ELSEIFB <аргумент2>
...
ELSE
...
ENDIF
```

Директива IFNB аналогична IFB, только обеспечивает ассемблирование, если 'аргумент' не пустой (не пробел):

```
IFNB <аргумент1>
...
ELSEIFNB <аргумент2>
...
ELSE
...
ENDIF
```

Директива IFDIF сопоставляет указанные 'аргумент1' и 'аргумент2', и обеспечивает ассемблирование при различии этих аргументов (с различием больших и маленьких букв). Угловые скобки (<>) должны указываться обязательно:

```
IFDIF <аргумент1>,<аргумент2>
...
ELSEIFDIF <аргумент3>,<аргумент4>
...
ELSE
...
ENDIF
```

Директива IFIDN сопоставляет указанные 'аргумент1' и 'аргумент2', и обеспечивает ассемблирование при идентичности этих аргументов (с различием больших и маленьких букв). Идентичность аргументов означает посимвольное совпадение 'аргумента1' и 'аргумента2':

```
IFIDN <аргумент1>,<аргумент2>
...
ELSEIFIDN <аргумент3>,<аргумент4>
...
ELSE
...
ENDIF
```

Есть еще две директивы выполняющие действия аналогичные директивам IFDIF и IFIDN, но без различия больших и маленьких букв:

`IFDIFI <аргумент1>,<аргумент2>` — сопоставляет указанные 'аргумент1' и 'аргумент2', и обеспечивает ассемблирование при различии этих аргументов (без различия больших и маленьких букв).

`IFIDNI <аргумент1>,<аргумент2>` — сопоставляет указанные 'аргумент1' и 'аргумент2', и обеспечивает ассемблирование при идентичности этих аргументов (без различия больших и маленьких букв).

Существуют условные директивы контроля ошибок, которые могут использоваться для отладки программ и контроля ошибок ассемблирования. Эти директивы можно помещать в критические точки программного кода и проверять условия ассемблирования в этих точках. Самой простой директивой контроля ошибок является .ERR, встретив ее, ассемблер прекратит работу с сообщением об ошибке. Пример:

```
IFDEF love
       ...
ELSE
       .ERR ; выводится ошибка, если имя love не определено
ENDIF
```

Другие условные директивы контроля ошибок:

- .ERR1 — ошибка только при первом проходе ассемблирования;
- .ERR2 — ошибка только при втором проходе ассемблирования;
- .ERRE выражение — ошибка, если выражение равно нулю (ложно);
- .ERRNZ выражение — ошибка, если выражение не равно нулю (истинно);
- .ERRDEF метка — ошибка, если метка определена;
- .ERRNDEF метка — ошибка, если метка не определена;
- .ERRB <аргумент> — ошибка, если аргумент пуст;
- .ERRNB <аргумент> — ошибка, если аргумент не пуст;
- .ERRDIF <арг1>,<арг2> — ошибка, если аргументы различны;
- .ERRDIFI <арг1>,<арг2> — ошибка, если аргументы различны (без различия больших и маленьких букв);
- .ERRIDN <арг1>,<арг2> — ошибка, если аргументы совпадают;
- .ERRIDNI <арг1>,<арг2> — ошибка, если аргументы совпадают (без различия больших и маленьких букв).

3.9. Макросы

Сейчас мы изучим языковое средство ассемблера MASM, из-за которого он собственно и получил свое название Macro Assembler (макроассемблер).

Макрос (макроопределение) — это символьное имя, заменяемое при ассемблировании на последовательность программных инструкций.

Обычно в качестве макросов оформляют повторяющиеся участки текста программы. Если в тексте программы будет встречаться вызов макроса (макрокоманда), то он будет заменяться на заранее определенные инструкции.

Отличие макроса от обычной подпрограммы заключается в том, что, если в исходном тексте будет несколько вызовов макроса, то его тело будет повторено столько же раз. Тело же обычной подпрограммы существует в единственном экземпляре независимо от количества вызовов. Хотя программа, использующая подпрограмму, расходует меньше места, но программа с макросом будет выполняться быстрее, т. к. в ней не будет лишних команд CALL и RET. Кроме того, тело макроса может изменяться в зависимости от тех аргументов, с которыми он вызван, обычная подпрограмма такое свойство обеспечить не может.

Макрос должен быть описан до первого вызова. Описание макроса еще называют макроопределением. Макроопределение начинается директивой MACRO и заканчивается ENDM. Как правило, макроопределения помещаются либо в начале исходного текста программы, либо в отдельный текстовый файл, который включается в исходную программу на этапе трансляции с помощью директивы INCLUDE.

Синтаксис макроопределения следующий:

```
Имя_макрокоманды MACRO Список_формальных_аргументов
        Тело макроопределения
        ENDM
```

Пример макроопределения без использования аргументов:

```
ah9  MACRO
        mov ah,9
        int  21h
        ENDM
```

Теперь в программе можно использовать имя ah9, как обычную инструкцию, а ассемблер заменит ее на две команды, содержащиеся в макроопределении.

Пример макроопределения с использованием аргументов:

```
addx  MACRO x1,x2,x3
        mov x1,x2      ;;
        add x1,x3      ;; сложение
        ENDM
```

Макрос addx складывает значения указанные в параметрах x2 и x3 и сохраняет результат в регистре, который должен быть указан в первом параметре x1. К примеру, если в программе с макроопределением вставить строку:

```
addx ax,20h,-7
```

то ассемблер заменит ее на следующие две инструкции:

```
mov ax,20h
add ax,-7
```

В макроопределении мы не случайно использовали комментарии, начинающиеся с двух точек с запятой (макрокомментарии). В отличие от обычных комментариев макрокомментарий не попадает в текст программы при подстановке макроса. Это увеличивает скорость ассемблирования программы с большим количеством макроопределений. Таким образом, в макроопределениях лучше всегда использовать только макрокомментарии.

3.9.1. Блоки повторений

В макросах могут создаваться блоки повторений с помощью специальных директив:

- Директивы WHILE и REPT

Данные директивы выполняют повторение участка программы заданное число раз.

Синтаксис директивы WHILE:

```
WHILE константное_выражение
инструкции
ENDM
```

Синтаксис директивы REPT:

```
REPT константное_выражение
инструкции
ENDM
```

Обе директивы повторяют инструкции столько раз, сколько это определено значением `константное_выражение`. Отличие директивы REPT от директивы WHILE в том, что она автоматически уменьшает на единицу значение `константное_выражение` после каждой итерации. В тело директивы WHILE нужно обязательно включать счетчик уменьшения значения `константное_выражение`, в ином случае цикл будет работать вечно, из-за чего на этапе транслирования ассемблер просто повиснет.

Пример макроса с использованием директивы WHILE:

```
mem0  MACRO  ln
       len=ln
       WHILE len
       db 0
       len=len-1
       ENDM
ENDM
```

Если в программе вставить строку:

```
mem0 100
```

то будет сгенерировано 100 директив DB 0.

Пример макроса с использованием директивы REPT:

```
mem1  MACRO  ln
       len=ln
       x=0
       REPT len
       db x
       x=x+1
       ENDM
ENDM
```

Если в программе вставить строку:

```
mem1 256
```

то будет сгенерировано 256 байт данных со значениями от 0 до 256.

- Директива IRP

Директива IRP делает процесс размножения участка программы более гибким, т. к. позволяет принимать параметры. Синтаксис директивы IRP:

```
IRP параметр,<значение1, значение2 ...>
инструкции
ENDM
```

Блок будет вызываться столько раз, сколько значений указано в списке (в угловых скобках). Обратите внимание угловые скобки (<>), обрамляющие список значений, являются обязательными. Например, выражение:

```
IRP reg,<ax,bx,cx,dx,si,di>
        push reg
ENDM
```

приведет к генерации следующих инструкций:

```
push ax
push bx
push cx
push dx
push si
push di
```

- Директива IRPC

Синтаксис директивы IRPC:

```
IRPC параметр,строка
инструкции
ENDM
```

Действие этой директивы подобно IRP, но отличается тем, что на каждой итерации параметр заменяется очередным символом из строки. Следовательно, количество повторений равно количеству символов в строке.

Если строка содержит пробелы, запятые и другие разделительные символы, то она должна заключаться в угловые скобки (<>). Обратите также внимание, в том случае если параметру в блоке повторения непосредственно предшествуют и/или следуют сразу за ним какие-либо символы, то параметр должен обрамляться (экранироваться) с соответствующих сторон символом амперсанда (&). В примере ниже параметр char заключен в одинарные кавычки, поэтому он обрамляется с обеих сторон амперсандами '&char&':

```
IRPC char,abcd
db '&char&',0
ENDM
```

Данный фрагмент эквивалентен следующей последовательности:

```
db 'a',0
db 'b',0
db 'c',0
db 'd',0
```

- Директива EXITM

Директива EXITM (сокращение от exit macro) выполняет немедленный выход из макроопределения или блока повторения.

- Директива LOCAL

Синтаксис директивы:

```
LOCAL метка
```

Данная директива заставляет транслятор сгенерировать уникальное имя метки при каждом вызове макрокоманды. Это обычно нужно для того, чтобы не возникало конфликта имен в том случае если метка с таким именем уже определена в основном тексте программы или при использовании макроса более одного раза. В данной директиве можно указывать сразу список меток через запятую.

Обычно метки, генерируемые транслятором, имеют следующий формат: ??nnnn, где вместо nnnn подставляется уникальный номер.

- Директива PURGE

Синтаксис директивы:

```
PURGE имя_макроса
```

Данная директива отменяет определенный ранее макрос.

ДЕНЬ 4

Основные команды ассемблера

Сегодня мы рассмотрим команды ассемблера, которые наиболее часто используются в программах на ассемблере.

4.1. Команды пересылки

Существует две основные команды пересылки – MOV и XCHG.

Мы уже неоднократно использовали команду MOV, настало время узнать о ней подробнее. Команда MOV имеет следующий синтаксис:

```
mov приемник,источник
```

С помощью команды MOV вы можете пересылать значение из источника в приемник. Несмотря на свое название "move" (от англ. — "перемещать"), команда на самом деле **копирует** значение из источника в приемник, а не перемещает.

Запомните: **ни одна машинная команда не способна манипулировать одновременно двумя операндами, находящимися в оперативной памяти**.

По этой причине возможны только следующие сочетания операндов в команде mov:

```
mov регистр,регистр   ; нельзя использовать одновременно два сегментных
                      ; регистра
mov регистр,память
mov память,регистр
mov регистр,непосредственный операнд ; нельзя использовать сегментный
                                     ; регистр
mov память,непосредственный операнд
```

Непосредственный операнд — это просто константа (число), которая может быть представлена именем (строкой, выражением), определенным с помощью операторов EQU или =.

Пересылка типа память-память невозможна. Исключениями в представленных сочетаниях также является то, что нельзя пересылать данные из одного сегментного регистра в другой и записывать непосредственный операнд в сегментный регистр — для этого нужно использовать промежуточные несегментные регистры.

Примеры:

```
mov ax,bx             ; mov регистр,регистр

mov es,cs             ; Ошибка! Нельзя использовать одновременно два
                      ; сегментных регистра.

mov eax,[00504032h]   ; mov регистр,память
mov [di],bx           ; mov память,регистр
mov [si],[80000h]     ; Ошибка! Нельзя пересылать из памяти в память.
```

```
mem db 5              ; Определение данного в памяти.
mov al,mem            ; Копирование из памяти в регистр.

mov ax,100            ; mov регистр,непосредственный операнд
mov ds,100            ; Ошибка! Нельзя в сегментный регистр пересылать
                      ; непосредственный операнд.
mov ds,ax             ; А вот сейчас правильно.
mov [si],64h          ; mov память,непосредственный операнд
```

Запись в квадратных скобочках, например [00504032h] означает, что нужно взять значение из памяти расположенное по адресу 00504032h. Напомню, что все ячейки памяти имеют уникальные адреса. На рис. 4.1 показан пример пронумерованных ячеек памяти с размещенными в этих ячейках некоторыми данными.

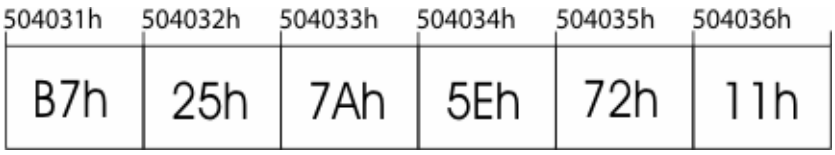

Рис. 4.1. Размещение данных в ячейках памяти

Согласно рисунку команда mov eax,[00504032h] поместит в регистр EAX двойное слово (32-разряда) из памяти начиная с адреса 504032h (по этому адресу расположено число 25h). После выполнения этой команды, регистр EAX будет содержать значение 725E7A25h, т. е. данные из памяти: 72h 5Eh 7Ah 25h . Данные в памяти хранятся в перевернутом виде согласно правилу "младший байт по младшему адресу", поэтому в EAX будет занесено именно значение 725E7A25h, а не 257A5E72h. Ассемблер берет из памяти значение размером в двойное слово, потому что EAX является 32-разрядным регистром, если бы первый операнд был 16-разрядным (AX), то из памяти было бы взято значение размеров в одно слово (16-разрядов), соответственно в 8-разрядный регистр (AH или AL) было бы помещено значение размером в 8-разрядов (один байт).

Если в команде не указывать квадратные скобки, например: mov eax,00504032h, то в EAX будет просто занесено значение 00504032h, т. к. без скобок, это просто непосредственный операнд (число).

С помощью оператора PTR (см. его описание в разд. 4.2) можно уточнить тип переменной, например можно переписать инструкцию выше следующим образом:

```
mov eax, dword ptr [00504032h]
```

В команде MOV можно также использовать регистр как ячейку памяти, если задействовать квадратные скобки:

```
mov eax, 403045h    ; Помещает в EAX значение 403045h.
mov dx, [eax]       ; Помещает в регистр DX значение размером в слово из
                    ; памяти по адресу указанному в регистре EAX [403045].
```

В данном случае ассемблер сам определяет, что из памяти нужно взять значение размером в слово (16 разрядов), так как dx является 16-разрядным регистром.

Важная деталь, в команде MOV приемник не может иметь такой вид: [00504032h]. Нужно обязательно использовать промежуточный регистр, например:

```
mov si,00504032h
mov [si],64h
```

В итоге по адресу 00504032h будет записано значение 64h.

Нужно также помнить еще об одной важной вещи, что размерность обоих операндов в команде MOV должна совпадать (либо байты, либо слова), иначе ассемблер зафиксирует ошибку. При пересылках никаких преобразований (байта в слово, слова в байт) не производится.

Примеры:

```
mov ebx,edx    ; правильно
mov cl,dl      ; правильно
mov bl,edx     ; ошибка! (BL – байт, EDX – двойное слово)
mov dx,al      ; ошибка! (DX – слово, AL – байт)
mov ah,257     ; ошибка! (AH – байт, 257 – больше байта)
mov bl, [edx] ; правильно, из памяти будет взято значение размером 8 бит.
```

Никакие флаги команда MOV не меняет.

Часто в программах необходимо переставлять местами какие-либо две величины, и хотя такую перестановку можно делать с помощью одних команд MOV, существует специальная команда для этого:

```
XCHG операнд1,операнд2
```

Примеры:

```
xchg ebx,eax                ; Обменять содержимое регистров EBX и EAX.
xchg ax, word ptr [di]      ; Обменять содержимое регистра AX и слова в
                            ; памяти по адресу в [DI].
xchg al,al                  ; Эта команда ничего не делает.
```

По аналогии с MOV команда XCHG не допускает перестановку типа память-память. Поэтому если нужна такая перестановка, то нужно задействовать промежуточный регистр. Например, для обмена значений двухбайтовых переменных X и Y можно поступить так:

```
mov al,x   ; AL=x
xchg al,y  ; AL=y, y=x
mov x,al   ; x=y
```

4.2. Оператор PTR

Оператор PTR используется для переопределения или уточнения типа метки или переменной. Этот оператор имеет следующий синтаксис:

тип PTR *выражение*

Тип может быть представлен именем или значением из табл. 4.1.

Таблица 4.1. *Типы оператора PTR*

Тип	Значение	Размер
BYTE	1	Байт
WORD	2	Слово (2 байта)
DWORD	4	Двойное слово (4 байта)
QWORD	8	Учетверенное слово (8 байт)
TBYTE	10	10 байт
NEAR	0FFFFh	Ближний переход
FAR	0FFFEh	Дальний переход

Типы BYTE, WORD, DWORD, QWORD и TWORD используются только с операндами памяти, а типы NEAR и FAR — только с метками.

Если оператор PTR не используется, то транслятор самостоятельно пытается определить тип ссылки или переменной, и, к сожалению, не всегда это делает правильно.

Примеры использования:

```
mov ax, dword ptr [0000]     ; Записать слово расположенное по адресу
                             ; ds:0000 в регистр AX.
mov word ptr Mem,15   ; Запись в ячейку памяти Mem размером в слово
                      ; десятичного числа 15.
```

```
mov word ptr [DI],0   ; Записать ноль как слово по адресу указанному в
                      ; регистре DI.
mov [DI], word ptr 0  ; Делает то же самое, что и предыдущий пример.
```

С помощью оператора PTR можно обращаться к отдельным байтам в многобайтовой переменной, пример:

```
.data
A dw 3612h            ; Двухбайтовая переменная 3612h.
.code
...
mov ah, byte ptr A    ; Пересылает первый байт (12h) в AH.
add ah, byte ptr A+1  ; Прибавляет второй байт (36h); в AH будет значение
                      ; 48h (12h+36h).
```

4.3. Способы адресации

Рассмотрев команды пересылки, мы можем рассмотреть все способы адресации операндов в командах на ассемблере.

Даже если вы поначалу не будете использовать все способы адресации, все равно я вам советую внимательно ознакомиться с ними, т. к. вы должны как минимум уметь читать чужой код с использованием различных методов адресации операндов.

4.3.1. Непосредственная адресация

Операнд указывается непосредственно в поле команды, например:

```
mov ax,312 ; здесь 312 задается непосредственно
```

4.3.2. Регистровая адресация

Операнд находится в одном из регистров. Например, оба операнда в команде

```
mov ds,bx
```

задаются с помощью регистрового способа адресации.

4.3.3. Косвенная адресация

Адрес операнда находится в одном из регистров — SI, DI, BX, BP. Например, команда

```
mov ax,[si]
```

помещает в регистр AX слово из ячейки памяти, смещение которой указано в регистре SI. Второй операнд задан с помощью косвенной адресации.

Следует отметить, что регистры SI, DI, BX, BP использовались до 80386 процессора, но в последующих поколениях процессоров для задания адреса операнда можно еще использовать регистры EAX, EBX, ECX, EDX, ESI, EDI, EBP и ESP.

4.3.4. Прямая адресация (адресация по смещению)

Абсолютный адрес операнда можно задавать в виде `СЕГМЕНТ:СМЕЩЕНИЕ`, где `СЕГМЕНТ` — адрес сегмента на который указывает какой-либо сегментный регистр (CS, DS, SS или ES), а `СМЕЩЕНИЕ` — адрес операнда относительно сегментного регистра (относительный адрес).

Примеры:

```
mov ax,ss:0037h
mov bx,es:var
```

4.3.5. Базовая адресация

Адрес операнда формируется сложением содержимого базового регистра (BP или BX) и смещения. Если смещение не задано, то предполагается нулевое значение.

Если используется BP, то адрес определенного операнда отсчитывается относительно сегмента, на который указывает регистр SS. Если используется BX, то адрес отсчитывается относительно сегмента, на который указывает регистр DS.

Например, команда:

```
mov ax,[bx+2]
```

помещает в регистр AX слово, которое находится в сегменте, указанном в DS, со смещением на два больше, чем число из BX.

Базовая адресация имеет множество альтернативных форм. Так, эквивалентными формами являются:

```
смещение[BP]
[смещение][BP]
[BP+смещение]
[BP].смещение
[BP]+смещение
```

В каждом случае адрес операнда равен сумме значений указанного смещения и содержимого заданного регистра.

4.3.6. Индексная адресация

Этот метод адресации подобен предыдущему, только адрес операнда формируется сложением содержимого индексного регистра (SI или DI) и смещения. Если смещение не задано, то предполагается нулевое значение.

В индексной адресации можно использовать множитель 1, 2, 4 или 8 чтобы прочитать элемент равный соответственно – байту, слову, двойному или учетверенному слову. Это называется масштабированием индексного регистра.

Примеры:

```
mov ax,[si]
mov ax,[di]
mov ax,12[di]
mov ax,[esi*4]+2
```

Индексная адресация имеет множество альтернативных форм. Так, эквивалентными формами являются:

```
смещение[SI]
[смещение][SI]
[SI+смещение]
[SI].смещение
[SI]+смещение
```

В каждом случае адрес операнда равен сумме значений указанного смещения и содержимого заданного индексного регистра.

4.3.7. Базовая-индексная адресация

Адрес операнда формируется сложением содержимого базового регистра (BP или BX) и индексного регистра (SI или DI), и смещения, если оно указано.

Если используется регистр BP, то адрес отсчитывается относительно сегмента, на который указывает регистр SS. В ином случае этот адрес отсчитывается относительно сегмента, на который указывает регистр DS.

Примеры:

```
mov ax,[bp][si]
mov ax,12[bp+di]
mov ax,[bx][si+2]
```

```
mov ax,2[bx][si]
```

Базово-индексная адресация имеет множество альтернативных форм. Так, эквивалентными формами являются:

```
смещение[BP][SI]
[смещение][BP][SI]
[BP+DI+смещение]
[BP+DI].смещение
[DI]+смещение+[BP]
```

В каждом случае адрес операнда равен сумме значений указанного смещения и содержимого заданных регистров. Можно комбинировать любой базовый регистр с любым индексным регистром, но комбинация двух базовых или двух индексных регистров не допускается.

4.3.8. Адресация по базе с индексированием и масштабированием

Эта общая схема адресации, в которую входят все случаи, рассмотренные нами ранее как частные. Адрес формируется как сумма смещения, базы и индекса, причем сумма может быть скорректирована с помощью масштабного множителя:

```
EA = База + (Индекс x Множитель) + Смещение
```

Схематически это показано на рис. 4.2.

$$CS:\ SS:\ DS:\ ES:\ FS:\ GS: \left[\begin{array}{l} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESP \\ EDI \\ ESI \end{array} + \begin{array}{l} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{array} * \begin{array}{l} 1 \\ 2 \\ 4 \\ 8 \end{array} + смещение \right]$$

Рис. 4.2. Полная форма адресации

4.4. Относительные операторы

Операторы сравнения сопоставляют `выражение_1` и `выражение_2`, после чего возвращают значение – единица, если условие выполняется, или ноль – если не выполняется.

Синтаксис операторов сравнения:

```
выражение_1 EQ выражение_2
выражение_1 NE выражение_2
выражение_1 LT выражение_2
выражение_1 GT выражение_2
выражение_1 GE выражение_2
```

В табл. 4.2 перечислены операторы сравнения вместе со значениями, которые они возвращают при выполнении условия.

Пример использования:

```
mov al,sum lt 15   ; если sum < 15, то al=1
```

Таблица 4.2. *Операторы сравнения*

Оператор	Возвращаемое значение
EQ	ИСТИНА, если выражения равны
NE	ИСТИНА, если выражения не равны
LT	ИСТИНА, если левое выражение меньше, чем правое
LE	ИСТИНА, если левое выражение меньше либо равно правому
GT	ИСТИНА, если левое выражение больше, чем правое
GE	ИСТИНА, если левое выражение больше или равно правому

4.5. Логические команды

Логические команды выполняют логические операции над битами, при этом бит 1 трактуется как "истина", а бит 0 — как "ложь". В ассемблере существует четыре основные логические команды: AND (операция логического И), OR (операция логического ИЛИ), XOR (операция логического "исключающего ИЛИ") и NOT (логическое отрицание):

```
AND приемник,источник
OR приемник,источник
XOR приемник,источник
NOT приемник
```

AND устанавливает бит результата в 1, если оба бита, бит источника и бит приемника установлены в 1.
OR устанавливает бит результата в 1, если один из битов, бит источника или бит приемника установлен в 1.
XOR устанавливает бит результата в 1, если бит источника установлен в 0.
NOT инвертирует бит источника.

В табл. 4.3 показаны все возможные варианты выполнения логических операций.

Таблица 4.3. *Логические операции*

И	ИЛИ	Исключающее ИЛИ	Отрицание
0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0	

Так как логические операции являются побитовыми, то для понимания их действия следует перевести числа в двоичный вид и провести операцию над каждым битом.

Примеры:

```
mov ax, 381h   ; ax = 0001110000001b
mov bx, 15EBh  ; bx = 1010111101011b
and ax,bx      ; ax = 0000110000001b  = 181h (результат)

mov ax, 20h  ; ax = 100000b
mov bx, 61h  ; bx = 1100001b
or ax,bx     ; ax = 1100001b  = 61h (результат)

mov ax, 33h  ; ax = 110011b
mov bx, 0Ah  ; bx = 001010b
xor ax,bx    ; ax = 111001b =  39h (результат)
```

```
mov eax, FFFF0000h ; eax = 11111111111111110000000000000000b
not eax ; eax = 00000000000000001111111111111111b = 0000FFFFh (результат)
```

4.6. Команды сдвига

Команды сдвига выполняют побитовый сдвиг в поле операнда вправо или влево, в зависимости от кода операции. Команды сдвига делятся на две основные группы:

- команды линейного (нециклического) сдвига;
- команды циклического сдвига.

Все команды сдвига устанавливают флаг переноса CF, причем последний выдвинутый бит становится значением флага переноса CF.

4.6.1. Команды линейного (нециклического) сдвига

Команды первой группы имеют следующий синтаксис:

```
SHR приемник,счетчик  ; логический сдвиг вправо
SHL приемник,счетчик  ; логический сдвиг влево
SAR приемник,счетчик  ; арифметический сдвиг вправо
SAL приемник,счетчик  ; арифметический сдвиг влево
```

Команды логического сдвига SHL и SHR сдвигают все биты операнда соответственно влево или вправо на столько бит, сколько указано в счетчике. При этом на место освобожденных разрядов вписываются нули.

Примеры:

```
mov al,57h ; al= 1010111b
shl al,4
; Теперь al = 1110000b. Флаг переноса cf=0, т. к. последний выдвинутый
слева бит был 0.


mov bh,5C ; bh= 1011100b
shr bh,3
; Теперь bh=0001011b. Флаг переноса  cf=1, т. к. последний выдвинутый
слева бит был 1.
```

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом обрабатывают знаковый разряд операнда.

Команда SAL такая же, как SHL, а вот команда SAR имеет одно принципиальное отличие от SHR. Команда SAR не обнуляет самый левые (старшие) биты после сдвига, как это делает SHR, а заполняет их значением самого старшего бита операнда, т. е. команда SAR сохраняет знак числа.

Примеры:

```
mov al,57h ; al= 1010111b
sal al,4
; Теперь al = 1110000b. Флаг переноса  cf=0, т. к. последний выдвинутый
слева бит был 0.


mov bl, B5 ; al = 10110101b
sar bl, 3
; Теперь al = 11110110b. Флаг переноса  cf=1, т. к. последний выдвинутый
слева бит был 1.


mov dl,26h ; dl = 00100110b
sar dl, 5
; Теперь dl = 0000001b. Флаг переноса  cf=0, т. к. последний выдвинутый
слева бит был 0.
```

Команды арифметического сдвига имеют одну уникальную особенность, которая часто используется программистами на ассемблере. Дело в том, что сдвиг влево

соответствует умножению операнда на степени двойки, а вправо, соответственно, делению операнда на степени двойки. То есть сдвиг влево на один бит соответствует умножению операнда на 2, сдвиг влево на два бита соответствует умножению на 4, на три бита — умножению на 8 и т. д.

Аналогично с делением. Сдвиг вправо на один бит соответствует делению на 2, сдвиг вправо на два бита соответствует делению на 4, на три бита — делению на 8 и т. д. Можете проверить это самостоятельно.

Команды арифметического сдвига выполняют умножение и деление на степень двойки намного быстрее, чем команды MUL, DIV и IDIV, поэтому программисты на ассемблере и используют их.

4.6.2. Команды циклического сдвига

Команды из второй группы имеют следующий синтаксис:

```
ROL операнд,счетчик   ; циклический сдвиг влево
ROR операнд,счетчик   ; циклический сдвиг вправо
RCL операнд,счетчик   ; циклический сдвиг влево через флаг переноса
RCR операнд,счетчик   ; циклический сдвиг вправо через флаг переноса
```

Команды циклического сдвига выполняют циклический сдвиг битов (ротацию) влево или вправо.

Команды ROL и ROR сдвигают все биты операнда соответственно влево или вправо на столько бит, сколько указано в счетчике, но в отличие от команд линейного (нециклического) сдвига не обнуляют освобожденные разряды, а снова вдвигают их с другой стороны, т. е. старшие биты попадают в младшие или наоборот.

Примеры:

```
mov al,C8h ; al= 11001000b
ror al,4
; Теперь al = 10001100b. Флаг переноса  cf=1, т. к. последний выдвинутый
слева бит был 1.


mov bh,D6h ; bh= 11010110b
rol bh,3
; Теперь bh = 10110110b. Флаг переноса  cf=0, т. к. последний выдвинутый
слева бит был 0.
```

Команды RCL и RCR выполняют действие аналогичное командам ROL и ROR, но включают флаг переноса CF в цикл, как если бы он был дополнительным битом в операнде. Последний выдвигаемый бит заносится в флаг CF, а значение CF при этом поступает в освободившуюся позицию Должно быть понятно, что команда RCL помещает значение CF в младший разряд, а команда RCR в старший.

Примеры:

```
mov al,C8h ; al= 11001000b
rcr al,4
; Теперь al = 11000110b. В старший разряд был вдвинут флаг переноса
cf=1, т. к. последний выдвинутый слева бит был 1.


mov bh,D6h ; bh= 11010110b
rcl bh,3
; Теперь bh = 01101100b. В младший разряд был вдвинут флаг переноса
cf=0, т. к. последний выдвинутый справа бит был 0.
```

4.7. Команды обработки строк/цепочечные команды

Команды обработки строк или цепочечные команды работают с последовательностями элементов. Один элемент может быть размером в байт (8 бит), слово (16 бит) или двойное слово (32 бита). Обычно последовательностями

элементов являются обычные строки символов, поэтому эти команды и называют командами обработки строк. Во всех цепочечных командах строка-источник должна находиться стандартно по адресу DS:SI (или DS:ESI), то есть в сегменте данных, определяемом регистром DS со смещением SI (или ESI), а строка-приемник по адресу в ES:DI (или ES:EDI) иначе говоря в сегменте данных, адресуемом сегментным регистром ES.

Все цепочечные команды за один раз обрабатывают только один элемент цепочки (строки). Для того чтобы команда выполнялась над всей строкой, используется один из префиксов повторения операций:

REP — повторять;

REPE — повторять, пока равно;

REPZ — повторять, пока ноль;

REPNE — повторять, пока не равно;

REPNZ — повторять, пока не ноль.

Эти префиксы указываются перед нужной цепочечной командой в поле метки и заставляют команду выполняться в цикле. Без префикса цепочечная команда выполняется один раз. Все префиксы анализируют значение регистра ECX/CX, а также флаг нуля ZF (кроме префикса REP).

Префикс повторения REP (от англ. REPeat — повторять) обычно используется совместно с командами MOVS и STOS и заставляет данные команды выполняться, пока содержимое в регистрах ECX/CX не станет равным 0. При этом цепочечная команда, перед которой стоит префикс, автоматически уменьшает содержимое ECX/CX на единицу. Та же команда, но без префикса этого не делает.

Префиксы повторения REPE (REPeat while Equal — повторять пока равно) и REPZ (REPeat while Zero — повторять пока ноль) являются абсолютно идентичными и обычно применяются с командами CMPS и SCAS для поиска отличающихся элементов цепочек. Эти префиксы заставляют цепочечные команды циклически выполняться, пока содержимое ECX/CX не равно нулю или пока флаг ZF не будет сброшен в 0, если это произойдет, управление будет передано следующей команде программы.

Префиксы повторения REPNE (REPeat while Not Equal — повторять пока не равно) или REPNZ (REPeat while Not Zero — повторять пока не ноль) также являются абсолютно идентичными и используются обычно с командами CMPS и SCAS, но для поиска совпадающих элементов. Эти префиксы заставляют цепочечные команды циклически выполняться до тех пор, пока содержимое ECX/CX не равно нулю или пока флаг ZF не будет установлен в 1, если это произойдет, управление будет передано следующей команде программы.

Строки в командах обработки строк могут обрабатываться либо от начала к концу, либо, наоборот, от конца к началу. Направление обработки задает флаг направления DF. Флаг направления анализируют цепочечные команды и если DF=0, то обрабатывают элементы в направлении возрастания адресов, если DF=1, то в направлении убывания адресов. Состоянием флага DF можно управлять с помощью команд CLD и STD, не имеющих операндов

CLD (Clear Dierction Flag) — сбрасывает флаг направления DF в 0, в результате чего при последующих строковых операциях значение в регистрах DI и SI будет увеличиваться.

STD (Set Direction Flag) — устанавливает флаг направления DF в 1, в результате чего при последующих строковых операциях значение в регистрах DI и SI будет уменьшаться.

Обратите внимание, что в каждой группе цепочечных команд (ниже) имеется одна команда с двумя операндами и несколько команд без операндов. На самом деле в процессоре существуют только цепочечные команды без операндов. При использовании команды с операндами транслятор ассемблера сам определяет по типу указанных операндов, какую из трех форм команды без операндов сгенерировать.

4.7.1. Команды пересылки цепочек

В эту группу входят следующие команды:

movs приемник,источник (MOVe String) — переслать цепочку;

movsb (MOVe String Byte) — переслать цепочку байт;

movsw (MOVe String Word) — переслать цепочку слов;

movsd (MOVe String Double word) — переслать цепочку двойных слов.

Команда MOVSB копирует байт из ячейки памяти, адресуемой парой регистров DS:SI (DS:ESI), в ячейку памяти, адресуемой парой регистров ES:DI (ES:EDI). Действие команды MOVSW аналогично, только копируется слово. Соответственно команда MOVSD копирует двойное слово. Команда MOVS с операндами транслируется в одну из трех команд без операндов: MOVSB, MOVSW, MOVSD.

Без префикса команда пересылки цепочек копирует только один элемент цепочки. С префиксом REP можно переслать до 64 Кбайт данных в 16-разрядной программе или до 4 Гбайт данных в 32-разрядной программе. Число пересылаемых элементов должно быть указано в регистре CX/ECX.

Алгоритм пересылки элементов цепочки может выглядеть так:

5. Загрузить адрес источника в регистр DS:SI (DS:ESI), а адрес приемника в ES:DI (ES:EDI).
6. Задать с помощью команды CLD или STD направление обработки цепочки (от начала к концу или от конца к началу).
7. Задать в регистре ECX/CX число обрабатываемых элементов.
8. Выдать команду MOVS (MOVSB, MOVSW или MOVSD) с префиксом REP.

Например, следующий блок кода копирует 10 байт строки text1 в строку text2:

```
            lea SI,text1
            lea DI,text2
            cld
            mov CX,10
      rep   movsb
```

При этом предполагается, что строка text1 находится в сегменте данных, на который указывает регистр DS, а строка text2 — в сегменте, на который указывает регистр ES (для программы типа .COM значение регистров DS и ES совпадает).

4.7.2. Команды сравнения цепочек

В эту группу входят следующие команды:

cmps приемник,источник (CoMPare String) — сравнить строки;

cmpsb (CoMPare String Word) — сравнить строку байт;

cmpsw (CoMPare String Word) — сравнить строку слов;

cmpsd (CoMPare String Double word) — сравнить строку двойных слов.

Команды этой группы сравнивают элементы цепочки-приемника (адресуемой регистрами ES:DI (ES:EDI)) с элементами цепочки-источника (адресуемой регистрами DS:SI (DS:ESI)). В зависимости от флага DF команда сравнения цепочек увеличивает или уменьшает адреса в регистрах SI и DI на один байт (CMPSB), слово (CMPSW) или двойное слово (CMPSD).

Принцип работы команд сравнения цепочек аналогичен команде сравнения CMP. Они так же, как и CMP, производят вычитание элементов, не записывая при этом результата, и устанавливают флаги ZF, SF и OF.

Если использовать команду сравнения цепочек с префиксами повторения REPNE/REPNZ или REPE/REPZ, то сравниваться будут столько элементов, сколько указано в регистре CX/ECX. При этом в случае префиксов REPNE/REPNZ сравнение прекратится при первом совпадении в цепочке, а в случае префиксов REPE/REPZ – при первом несовпадении.

Для определения причины, которая привела к выходу из цикла сравнения цепочек, обычно используется команда условного перехода JCXZ. Эта команда анализирует содержимое регистра CX/ECX, и если оно равно нулю передает управление на метку, указанную в качестве операнда в JCXZ. Если значение в CX/ECX не равно нулю, то это означает, что выход произошел по причине совпадения или несовпадения очередных элементов в цепочке.

В листинге 4.1 в качестве примера показана программа сравнения двух строк. Если строки text1 и text2 равны, то осуществляется переход на метку Equality.

Листинг 4.1. Программа сравнения двух строк (cmpstr.asm)

```
        .model small
        .stack 100h
        .code
start:
        mov     ax,@data
        mov     ds,ax
        mov     es,ax

        lea     si,text1        ; в SI адрес начала строки text1
        lea     di,text2        ; в DI адрес начала строки text2
        cld

        ; в CX длина строки text1 (число символов для сравнения)
        mov     cx,len_text1

        repe cmpsb              ; сравнивать, пока элементы равны
        je      Equality

        mov     dx,offset mes_no
        mov     ah,9
        int     21h

        jmp     exit

Equality:
        mov     dx,offset mes_yes
        mov     ah,9
        int     21h
exit:                           ; выход из программы
        mov     ax,4C00h
        int     21h

        .data
text1           db      "Ivan Sklyaroff"        ; первая строка
len_text1=$-text1                               ; длина первой строки
text2           db      "Ivan Sklyaroff"        ; вторая строка
mes_yes         db      "Строки одинаковы",'$'
mes_no          db      "Строки различны",'$'

        end     start
```

4.7.3. Команды сканирования цепочек

В эту группу входят следующие команды:

scas приемник (SCAning String) — сканировать цепочку;

scasb (SCAning String Byte) — сканировать цепочку байт;

scasw (SCAning String Word) — сканировать цепочку слов;

scasd (SCAning String Double Word) — сканировать цепочку двойных слов.

Команды сканирования цепочек осуществляют поиск заданного значения в строке. Значение, которое требуется найти, необходимо предварительно поместить в регистр AL (если ищется байт), в AX (если ищется слово) или в EAX (если осуществляется поиск двойного слова), а адрес строки должен быть сформирован в регистре ES:DI (ES:EDI). Принцип поиска тот же, что и в команде сравнения CMPS, то есть выполняется последовательное вычитание (из содержимого регистра аккумулятора содержимое очередного элемента цепочки) и в зависимости от результатов вычитания производится установка флагов, при этом ни один из операндов не изменяется.

Если использовать команду сканирования цепочки с префиксом REPE или REPZ, то команда будет выполняться в цикле до тех пор, пока не будет достигнут конец цепочки (содержимое CX/ECX равно 0) или пока не встретится элемент, отличный от элемента в регистре AL/AX/EAX.

Если применить префикс REPNE или REPNZ, то цикл будет выполняться пока в цепочке не встретится элемент, совпадающий с элементом в регистре AL/AX/EAX или пока не будет достигнут конец цепочки (содержимое CX/ECX равно 0).

То есть команда сканирования цепочки с префиксом REPE или REPZ позволяет найти элемент цепочки, отличающийся от заданного в аккумуляторе, а с префиксом REPNE или REPNZ – совпадающий по значению с элементом в аккумуляторе.

В листинге 4.2 в качестве примера показана программа, которая ищет символ 't' в строке. Если символ найден, то осуществляется переход на метку found.

Листинг 4.2 Поиск символа в строке (scastr.asm)

```
        .model small
        .stack 100h
        .code
start:
        mov     ax,@data
        mov     ds,ax
        mov     es,ax

        cld
        lea     di,text1        ; в DI адрес начала строки text1
        mov     cx,len_text1    ; в CX длину строки text1

        mov     ah,0
        mov     al,'t'          ; ищем символ 't' в строке
        repne   scasb
        je      found

        mov     dx,offset mes_no
        mov     ah,9
        int     21h

        jmp     exit
found:
        mov     dx,offset mes_yes
        mov     ah,9
        int     21h
exit:
        mov     ax,4C00h
        int     21h

        .data
```

```
text1          db      "Not many people can do it."
len_text1=$-text1
mes_yes        db      "Символ найден",'$'
mes_no         db      "Символ не обнаружен",'$'

        end     start
```

4.7.4. Команды загрузки элемента из цепочки в аккумулятор

В эту группу входят следующие команды:

lods источник (LOaD String) — загрузить элемент из цепочки в регистр-аккумулятор AL/AX/EAX;

lodsb (LOaD String Byte) — загрузить байт из цепочки в регистр AL;

lodsw (LOaD String Word) — загрузить слово из цепочки в регистр AX;

lodsd (LOaD String Double Word) — загрузить двойное слово из цепочки в регистр EAX.

Данные команды позволяют извлечь элемент из цепочки и поместить его в регистр-аккумулятор AL, AX или EAX. Адрес строки должен быть сформирован в регистрах ES:DI (ES:EDI). Размер извлекаемого элемента определяется применяемой командой. Так как за одно выполнение команда загрузки элемента из цепочки занимает регистр-аккумулятор, то практической пользы от использования префиксов повторения в данном случае нет. Часто команда загрузки элемента из цепочки полностью соответствует команде MOV, хотя MOV генерирует 3 байта машинного кода, а LODS – только один, но требует инициализации регистра ES:DI (ES:EDI).

Команды, эквивалентные команде LODSB:

```
MOV  AL,[SI]
INC  SI
```

Пример использования команды lodsb, совместно с инструкцией stosb (см. ниже) показан в листинге 4.3. Данная программа переписывает строку text2 символами строки text1 и выводит новое содержимое text2 на экран.

4.7.5. Команды переноса элемента из аккумулятора в цепочку

В эту группу входят следующие команды:

stos приемник (STOre String) — сохранить элемент из регистра-аккумулятора AL/AX/EAX в цепочке;

stosb (STOre String Byte) — сохранить байт из регистра-аккумулятора AL/AX/EAX в цепочке;

stosw (STOre String Word) — сохранить слово из регистра AX в цепочке;

stosd (STOre String Double Word) — сохранить двойное слово из регистра EAX в цепочке.

Данные команды осуществляют действия обратные команде LODS, то есть извлекают значение из регистра-аккумулятора и помещают его в элемент цепочки. Адрес строки должен адресоваться регистрами ES:DI (ES:EDI).

Префикс повторения с командами переноса элемента из аккумулятора в цепочку обычно используются для заполнения области памяти какими-либо фиксированными значениями.

Пример использования команды stosb показан в листинге 4.3. Данная программа переписывает строку text2 символами строки text1 и выводит новое содержимое text2 на экран.

Листинг 4.3. Замена символов строки (stostr.asm)

```
        .model small
```

```
        .stack 100h
        .code
start:
        mov     ax,@data
        mov     ds,ax
        mov     es,ax

        cld
        lea     si,text1        ; в SI адрес начала строки text1
        lea     di,text2        ; в DI адрес начала строки text2
        mov     cx,len_text1    ; в CX длина строки text1

Mark:   lodsb                   ; очередной элемент из text1 в AL
        stosb                   ; очередной элемент из AL в text2
        loop    Mark

        mov     dx,offset text2 ; выводим строку-приемник
        mov     ah,9
        int     21h

        mov     ax,4C00h
        int     21h

        .data
text1           db      "Пересылаемая строка$"
len_text1=$-text1
text2           db      "Строка-приемник$"

        end     start
```

4.7.6. Команды ввода элемента цепочки из порта ввода-вывода

В эту группу входят следующие команды:

ins приемник,порт (Input String) — ввести элементы из порта ввода-вывода в цепочку;

insb (INput String Byte) — ввести из порта цепочку байт;

insw (INput String Word) — ввести из порта цепочку слов;

insd (INput String Double Word) — ввести из порта цепочку двойных слов.

Данные команды считывают данные из порта ввода-вывода, номер которого должен быть указан в регистре DX, байт (INSB), слово (INSW) или двойное слово (INSD) в память по адресу ES:EDI (или ES:DI, в зависимости от разрядности адреса). После пересылки команда в зависимости от флага направления DF увеличивает (DF=0) или уменьшает (DF=1) содержимое регистра DI/EDI на величину, равную размеру элемента, участвовавшего в операции пересылки.

Команда INS в зависимости от типа указанного операнда транслируется в одну из трех форм этой команды без операндов (INSB, INSW или INSD).

```
.data
mem db 100 dup(" ")
.code
...
        push ds
        pop es
```

```
        mov dx,777h
        lea di,mem
        mov cx,100
rep insb
...
```

4.7.7. Команды вывода элемента цепочки в порт ввода-вывода

В эту группу входят следующие команды:

outs порт,источник (Output String) — вывести элементы из цепочки в порт ввода-вывода;

outsb (OUTput String Byte) — вывести цепочку байт в порт ввода-вывода;

outsw (OUTput String Word) — вывести цепочку слов в порт ввода-вывода;

outsd (OUTput String Double Word) — вывести цепочку двойных слов в порт ввода-вывода.

Данные команды записывают данные в порт ввода-вывода, номер которого должен быть указан в регистре DX, байт (OUTSB), слово (OUTSW) или двойное слово (OUTSD) из памяти по адресу DS:ESI (или DS:SI, в зависимости от разрядности адреса). После пересылки команда в зависимости от флага направления DF увеличивает (DF=0) или уменьшает (DF=1) содержимое регистра SI/ESI на величину, равную размеру элемента, участвовавшего в операции пересылки.

Команда OUTS в зависимости от типа указанного операнда транслируется в одну из трех форм этой команды без операндов (OUTSB, OUTSW или OUTSD).

В качестве примера ниже показан фрагмент программы, который выводит последовательность символов в порт с номером 340h:

```
.data
str db "Текст для вывода в порт"
len_str=$-str
.code
...
        mov dx,340h
        lea di,str
        mov cx,len_str
rep outsb
...
```

4.8. Команды работы с адресами и указателями

В эту группу входят следующие команды:

lea приемник,источник — загрузка эффективного адреса;

lds приемник,источник — загрузка указателя в регистр сегмента данных DS;

les приемник,источник — загрузка указателя в регистр дополнительного сегмента данных ES;

lgs приемник,источник — загрузка указателя в регистр дополнительного сегмента данных GS;

lfs приемник,источник — загрузка указателя в регистр дополнительного сегмента данных FS;

lss приемник,источник — загрузка указателя в регистр сегмента стека SS.

Данные команды также как команда MOV копируют источник в приемник, только в отличие от MOV копируют не данные, а эффективный адрес данных (то есть смещение данных относительно начала сегмента данных).

Таким образом, источник должен быть ссылкой на ячейку памяти, а в качестве приемника может выступать любой 16-битный регистр, кроме сегментных.

Команда

```
lea BX,string
```

эквивалентна команде

```
mov BX,offset string
```

Но команда LEA имеет преимущество по сравнению с использованием оператора offset в команде MOV в том, что источник может содержать индексы. Например, следующая команда будет работать без ошибок:

```
lea BX,string[SI]
```

в то время как команда

```
mov BX,offset string[SI]
```

вызовет ошибку на этапе ассемблирования.

4.9. Команды трансляции (преобразования) по таблице

На самом деле речь здесь пойдет только об одной команде, которая имеет две формы записи:

```
xlat адрес
xlatb
```

Эта команда преобразует байт согласно таблице преобразований, поэтому часто используется для шифрования методом подстановки.

Команда заменяет значение в регистре AL другим байтом из таблицы в памяти, начальный адрес которой находится в регистре BX. Таблица в памяти — это просто строка байт размером от 1 до 255 байт.

Обратите внимание: аргумент XLAT на самом деле не используется процессором и может выполнять лишь роль комментария. Если комментарий не нужен, можно использовать форму записи XLATB.

Адрес таблицы предварительно должен быть загружен в регистр BX (например, с помощью команды LEA). В AL предварительно должен быть помещен байт, который нужно перевести. Адрес байта в таблице, которым будет производиться замещение содержимого AL, определяется суммой (BX)+(AL), таким образом, первоначальное значение в AL выполняет роль индекса (смещения) в таблице преобразований.

Пример использования:

```
lea bx, HEX_TABLE     ; адрес таблицы - в BX
mov al,7              ; переводимая цифра - в AL
xlatb                 ; преобразование
; таблица преобразования в сегменте данных
HEX_TABLE      db     '0123456789ABCDEF'
```

После преобразования в регистре AL будет находиться символ '7'.

ДЕНЬ 5

Арифметические команды. Сопроцессор

В этот день мы с вами научимся выполнять все арифметические операции над числами на ассемблере. Кроме простейших арифметических действий, таких как сложение и вычитание научимся также вычислять синусы, косинусы, квадратные корни и пр. Мы не будет далеко лезть в математические дебри, но при соответствующей математической подготовке после прочтения материала этого дня вы без особого труда сможете выполнять любые сложные математические операции на ассемблере, например, вычисление интегралов и решение дифференциальных уравнений.

Сначала мы рассмотрим работу с целыми, а затем с вещественными числами (с фиксированной и плавающей точкой). Если вы уже подзабыли математику и не помните, что такое вещественные числа, плавающая точка и пр. не беспокойтесь, я популярно объясню все необходимые основы математики.

Как правило, целочисленные операции выполняет центральный процессор, а операции с плавающей точкой возлагаются на сопроцессор (FPU/NPX). На первом дне я уже говорил, что, начиная с процессора 80486DX, сопроцессор располагается на одном кристалле с центральным процессором; до процессора 80486DX сопроцессор представлял собой отдельную микросхему на материнской плате. Подробнее с сопроцессором и его командами мы еще познакомимся на этом дне при изучении операций с числами с плавающей точкой.

Что касается вещественных чисел с фиксированной точкой, то операции с ними на ассемблере выполняются точно также как с целыми числами, а потому их выполнение обычно возлагается на центральный процессор.

Операции с плавающей точкой можно выполнять и без сопроцессора, но без привлечения команд сопроцессора это будет сложнее и потребует больших вычислительных затрат, а значит приложение будет выполняться медленнее (причем значительно). Сопроцессор тоже способен выполнять целочисленные операции, но делает он это медленнее центрального процессора.

5.1. Арифметические операторы

Выполнение простейших арифметических действий на ассемблере можно выполнять с помощью обычных арифметических операторов: + (плюс), - (минус), * (умножение), / (целочисленное деление), MOD (остаток от деления). К арифметическим операторам относят также унарный плюс (+) и унарный минус (-), которые соответственно могут ставиться перед положительными и отрицательными числами. В выражениях могут использоваться также круглые скобки **()**. Часть выражения, заключенная в круглые скобки, вычисляется в первую очередь.

Примеры использования арифметических операторов:

```
db 3+2*5              ; db 13
db (8+3)*2            ; db 22
mov al,14 / 4         ; mov al,3
mov al,14 mod 4       ; mov al,2
mov al,10 - +5        ; mov al,5
mov al,10 - -5 ; mov al,15
```

5.2. Команды выполнения целочисленных операций

Числа, не имеющие запятой, называются целыми числами. Целые числа могут быть со знаком и без знака. Очень просто на обычном листке бумаги указать знак у числа ("+" или "-"), но не так просто это сделать в двоичном представлении числа. Как к двоичному числу "приписать" знак будет рассказано ниже.

Все команды целочисленных операций можно разделить на две группы:

- команды, работающие с целыми двоичными числами;
- команды, работающие с целыми BCD-числами (двоично-десятичными).

Что такое двоичные числа вы уже знаете. Что касается BCD-чисел, то это особый формат представления привычных нам десятичных чисел в двоичном виде. Вспомните, как не просто осуществляется перевод чисел из десятичной системы счисления в двоичную и обратно (см. разд. 1.2). Как человеку не просто выполнять эти преобразования, также не просто выполнять их и компьютеру. Для того, чтобы уменьшить затраты на преобразование чисел из десятичной системы счисления в двоичную, и наоборот, был придуман двоично-десятичный формат (binary-coded decimal, BCD). Ниже мы подробнее познакомимся с BCD-числами. Даже если вы не будете использовать BCD-числа в своих программах, познакомиться с ними необходимо, т. к. BCD-числа часто используются в компьютере, например в BCD-формате хранится информация о текущем времени в КМОП-микросхеме компьютера.

Думаю понятно, что команды целочисленных операций, предназначенные для работы с обычными двоичными числами, будут неправильно работать с BCD-числами и наоборот, поэтому я и разделил команды выполнения целочисленных операций на две группы. Но прежде чем рассмотреть эти две группы команд, узнаем, что такое целые двоичные числа и BCD-числа.

5.2.1. Целые двоичные числа

Целые двоичные числа без знака это обычные числа в двоичной системе счисления. Диапазоны значений целых чисел без знака:

- байт: от 0 до 255;
- слово: от 0 до 65 535;
- двойное слово: от 0 до 4 294 967 295.

Для указания знака в двоичном числе используется самый старший бит: бит 1 соответствует знаку "—", а бит 0 знаку "+". Но все не так просто. Например, если взять два двоичных числа, отличающиеся только старшим битом: 00110010 и 10110010. То их сумма как отрицательного и положительного числа должна по идее давать 0. Но если вы сложите эти числа, то получите 11100100 = 228.

Для решения этой проблемы поступают следующим образом: положительные двоичные числа записываются как обычные двоичные беззнаковые числа со старшим битом равным 0, а для представления отрицательных чисел используют так называемый "дополнительный код". Для этого выполняют инверсию положительного числа, то есть заменяют в двоичном числе все единицы на нули, а нули на единицы, затем к полученному результату прибавляют 1.

Например, получим дополнительный код из числа 00110010 = 50:

00110010

11001101 + 1 = 11001110

Проверим, что дополнительный код является числом -50: сумма с 50 должна быть равна нулю:

+50 + (-50) = 00110010 + 11001110 = 100000000

Единица в 9-м разряде не помещается в байт, поэтому мы действительно получили 0.

Диапазоны значений целых чисел со знаком, имеющих разную длину:

- байт: от -128 до +127;
- слово: от -32 768 до +32 767;

- двойное слово: от -2 147 483 648 до +2 147 483 647.

5.2.2. BCD-числа

В BCD-формате каждая десятичная цифра числа кодируется группой из четырех бит.

Существует две разновидности BCD-формата: упакованный и неупакованный форматы.

В неупакованном формате в одном байте (в младших четырех битах) размещается одна десятичная цифра, значение старшей половины байта в этом случае не определено. Пример представления беззнакового десятичного числа в неупакованном BCD-формате:

$$5096 = \underbrace{xxxx0101}_{1байт} \quad \underbrace{xxxx0000}_{2байт} \quad \underbrace{xxxx1001}_{3байт} \quad \underbrace{xxxx0110}_{4байт}$$

В упакованном формате в каждом байте хранится две цифры числа (в старшей и младшей половине). Пример представления беззнакового десятичного числа в упакованном BCD-формате:

$$5096 = \underbrace{01010000}_{1байт} \quad \underbrace{10010110}_{2байт}$$

BCD-числа со знаком образуются почти также как обычные двоичные числа со знаком. Положительные BCD-числа записываются как беззнаковые, но знак хранится и обрабатывается в отдельном байте, который заполнен нулями. Отрицательные BCD-числа представляются в дополнительном коде по следующему алгоритму:

— все цифры десятичного числа, кроме младшей (самой правой), заменяются на их дополнение до 9 (т. е. 9 – {цифра});

— последняя (младшая) цифра заменяется дополнением до 10 (т. е. 10 – {цифра});

— знак хранится и обрабатывается в отдельном байте, который заполнен нулями, кроме старшего бита, который для отрицательного числа устанавливается в 1.

Например, получим дополнительный код из числа 152 (т. е. отрицательное число -152) в упакованном BCD-формате:

```
0152 = 0000 0001 0101 0010
дополнительный код (-0152) = 1000 0000 1001 1000 0100 1000
```

Способ определения и обработки BCD-чисел в программе целиком зависит от предпочтения программиста. Обычно неупакованные BCD-числа определяются в программе с помощью директивы db (на каждое число отводится один байт), например:

```
bcd_unpack db 5,9,0,1,4 ; неупакованное BCD-число 59014
```

Часто программисты для удобства обработки определяют неупакованные BCD-числа в обратном порядке (в этом случае в памяти компьютера число будет представлено в прямом порядке), пример:

```
bcd_unpack db 4,1,0,9,5 ; неупакованное BCD-число 59014
```

Кроме того, часто используются ASCII-коды для представления цифр неупакованного BCD-числа для удобства вывода на экран:

```
bcd_unpack db '59014' ; неупакованное BCD-число '5','9','0','1','4'
```

или в обратном порядке:

```
bcd_unpack db '41095' ; неупакованное BCD-число '5','9','0','1','4'
```

Упакованные BCD-числа обычно определяются с помощью директивы dt, например:

```
bcd_pack dt 325417 ; упакованное BCD-число 325417
```

5.2.3. Команды, работающие с целыми двоичными числами

5.2.3.1. Сложение и вычитание

Основной инструкцией для сложения целых двоичных чисел является ADD, а для вычитания SUB.

Инструкция ADD складывает операнд-источник с операндом-приемником и записывает результат в операнд-приемник:

```
ADD приемник,источник
```

Инструкция SUB вычитает операнд-источник из операнда-приемника и записывает результат в операнд-приемник:

```
SUB приемник,источник
```

Пример:

```
mov ax,10
add ax,6 ; ax=ax+6
sub ax,9 ; ax=ax-9
```

В результате выполнения этих инструкций в регистре AX окажется число 7.

5.2.3.2. Инкремент и декремент

В ассемблере, как и в языках Си/Си++, имеются операторы инкремента и декремента:

```
inc операнд
dec операнд
```

Инкремент увеличивает значение операнда на 1, а декремент уменьшает значение операнда на 1, т.е. команда INC аналогична команде ADD приемник,1, а команда DEC аналогична команде SUB приемник,1. Единственное отличие: команды INC и DEC не меняют флаг переноса CF.

Преимущество команд INC и DEC в том, что они занимают меньше места в памяти и выполняются быстрее, чем соответствующие команды ADD и SUB.

Примеры:

```
mov al,15     ; al=15
inc al        ; al=16
dec al        ; al=15
```

5.2.3.3. Умножение и деление

Для умножения чисел без знака используется инструкция MUL, а для умножения чисел со знаком инструкция — IMUL:

```
MUL операнд-источник
IMUL операнд-источник
```

Обратите внимание, что данные команды требуют указывать только один операнд. В качестве второго операнда команды MUL и IMUL используют либо содержимое регистра AL при операциях над байтами, либо AX при операциях над словами, либо EAX при операциях над двойными словами. Результат умножения соответственно помещается в AX, в DX:AX или в EDX:EAX.

Выбор подходящей команды умножения и контроль за форматом обрабатываемых чисел лежит на самом программисте.

Деление чисел без знака выполняет команда DIV, а чисел со знаком — команда IDIV:

```
DIV операнд-источник
IDIV операнд-источник
```

Также как инструкции умножения, инструкции деления требуют указывать только один операнд. Операнд-источник является делителем, а в качестве делимого используется либо содержимое регистра AL при операциях над байтами, либо AX при операциях над словами, либо EAX при операциях над двойными словами. Результат деления помещается в AL, AX или EAX, а остаток – в AH, DX или EDX соответственно.

Также как в случае команд умножения выбор подходящей команды деления и контроль за форматом обрабатываемых чисел лежит на самом программисте.

5.2.3.4. Изменение знака числа

Команда NEG изменяет знак числа на противоположный:

```
NEG операнд
```

Физически команда просто вычитает операнд из нуля (`операнд = 0 - операнд`). При этом выполняются все правила представления чисел со знаком, о которых мы говорили выше, т. е. числа конвертируются в двоичный дополнительный код.

Команда NEG изменяет состояние следующих флагов: CF (флаг переноса), ZF (флаг нуля), SF (флаг знака), OF (флаг переполнения), AF (флаг служебного переноса), PF (флаг четности). Эти флаги можно использовать для анализа полученного результата.

Команду NEG удобно использовать для получения абсолютного значения числа следующим образом:

```
Mark:  neg AX
       jl Mark
```

Если знак числа отрицательный будет выполнен переход на первую команду еще раз.

5.2.4. Ввод и вывод чисел

При работе с числами на ассемблере перед каждым программистом встает вопрос, как осуществлять ввод и вывод чисел в программе? Действительно, ввод осуществляется в ASCII-кодах, а вычисления нужно производить с обычными числами; результат вычислений на экран вновь нужно выводить в ASCII-кодах.

Таким образом, при работе с числами в программе на ассемблере необходимы две процедуры: первая должна преобразовывать введенные пользователем числа из ASCII-формата в бинарные числа, а вторая процедура должна наоборот преобразовывать результат вычислений в ASCII-коды для вывода на экран. К сожалению, стандартных команд, которое бы это делали, не существует. В листинге 5.1 вы можете видеть программу, которая запрашивает два числа, складывает их и выводит результат на экран.

В этой программе процедура str2int преобразовывает каждое число из буфера в ASCII-формате в соответствующие бинарное число.

Другая процедура int2str преобразовывает бинарное число (результат вычисления) в число в системе счисления base в ASCII-формате.

Обращаю ваше внимание, что авторство на данные процедуры принадлежит не мне (если авторы объявятся, я укажу их в переиздании этой книги).

К настоящему времени весь код программы вы должны в состоянии понять самостоятельно, а комментарии вам в этом помогут.

Ассемблирование и линковка выполняется обычным образом:

```
ml numascii.asm
```

Листинг 5.1. Программа, демонстрирующая ввод и вывод чисел (numascii.asm)

```
        .model small
        .stack 100h

        .data
mess1   db "Введите число #1: ",'$'
mess2   db "Введите число #2: ",'$'
mess3   db "Результат: ",'$'
mess4   db "Ошибка ввода $"
crlf    db 0Dh,0Ah,'$'          ; перевод строки
n       db 5                    ; максимальный размер буфера ввода
nlength db 0                    ; здесь будет размер буфера после считывания
ncontents db 5 dup (?)          ; содержимое буфера

buff    db 100 dup(0),'$'       ; размер буфера для преобразования числа
table1 db '0123456789ABCDEF'     ; таблица соответствия
base    dw 10                    ; указываем систему счисления в которой
                                ; желаем выводить числа
                                ; например: 10 - десятичная, 2 -двоичная,
```

```
                              ; 16 - шестнадцатеричная
        .code
; процедура вывода строки на экран
output PROC
        mov     ah,9
        int     21h
        ret
output ENDP

; процедура считывания введенной строки,
; которая помещается в буфер ncontents
input PROC
        mov     ah,0Ah
        int     21h
        ret
input ENDP

; процедура преобразования числа в ASCII-формате
; из буфера ncontents в соответствующее бинарное число,
; результат преобразования помещается в AX
str2int PROC
        xor     di,di           ; DI указывает номер байта в буфере
        xor     ax,ax
        mov     cl,nlength
        xor     ch,ch
        xor     bx,bx
        mov     si,cx           ; в SI длина буфера
        mov     cl,10           ; множитель для MUL
next1:
        mov     bl,byte ptr ncontents[di] ; взять байт из буфера
        sub     bl,'0'          ; вычитаем ASCII-код символа '0'
        jb      error1          ; если код символа меньше, чем код '0'
        cmp     bl,9            ; или больше, чем '9'
        ja      error1          ; выходим из программы с сообщением об
ошибке
        mul     cx              ; умножаем значение в AX на 10
        add     ax,bx           ; добавляем к нему новую цифру
        inc     di              ; увеличиваем счетчик
        cmp     di,si
        jb      next1
        ret
error1:
        mov     dx,offset mess4
        mov     ah,9
        int     21h
        jmp     exit
str2int ENDP

; процедура вывода на экран числа в системе счисления base
; число передается в процедуру в регистре AX, а
; адрес результата помещается в DX
int2str PROC
```

```
; сначала очищаем буфер (заполняем нулями)
       xor    di,di
       mov    cx,99
zeroizing:
       mov    byte ptr buff[di],0
       inc    di
loop zeroizing

; заносим в di адрес последнего байта буфера,
; т. к. буфер будет заполняться от конца к началу
       xor    di,di
       mov    di,offset buff+99

; цикл получения цифр числа путем вычисления остатков
; от деления на основание системы
next2: xor    dx,dx
       div    base
       mov    si,offset table1
       add    si,dx
       mov    dl,[si]
       mov    [di],dl
       dec    di
       cmp    ax,0          ; если AX не равно 0 повторяем цикл
       jnz    next2
       mov    dx,di         ; в DX заносим адрес результата
       ret
int2str ENDP

start:  mov    ax,@data
       mov    ds,ax

       mov    dx,offset mess1      ; запрос первого числа
       call   output

       mov    dx,offset n          ; принимаем ввод первого числа
       call   input

       mov    dx,offset crlf       ; перевод строки
       call   output

       call   str2int              ; преобразование ASCII-строки в число
                                   ; (результат помещается в AX)

       push   ax                   ; сохраняем первое число в стек

       mov    dx,offset mess2      ; запрос второго числа
       call   output

       mov    dx,offset n          ; принимаем ввод второго числа
       call   input
```

```
        call    str2int             ; преобразование ASCII-строки в число
                                    ; (результат помещается в AX)

        pop     bx              ; восстанавливаем первое число из стека
        add     ax,bx           ; складываем числа
        push    ax              ; сохраняем результат в стек

        mov     dx,offset crlf      ; перевод строки
        call    output

        mov     dx,offset mess3     ; сообщение о выводе результата
        call    output

        pop     ax                  ; восстанавливаем результат из стека
        call    int2str             ; преобразовываем в ASCII-строку
        call    output              ; выводим результат на экран

exit:                               ; выходим из программы
        mov     ax,4C00h
        int     21h

        end     start
```

5.2.5. Команды, работающие с целыми BCD-числами

5.2.5.1. Сложение и вычитание неупакованных BCD-чисел

Выше я уже говорил, что команды целочисленных операций, предназначенные для работы с обычными двоичными числами, будут неправильно работать с BCD-числами и наоборот. Пример сложения двух неупакованных BCD-чисел 5 и 6 с помощью команды ADD:

5 = 0000 0101

6 = 0000 0110

11 = 0000 1011

Однако в BCD-формате число 11 должно быть представлено как 0000 0001 0000 0001.

Разработчики микропроцессоров решили не вводить специальные команды для работы с BCD-числами, а ввести только несколько корректировочных команд.

Для корректировки сложения и вычитания неупакованных BCD-чисел имеются две команды:

- AAA (ASCII Adjust for Addition) — коррекция результата сложения для представления в символьном виде;
- AAS (ASCII Adjust for Subtraction) — коррекция результата вычитания для представления в символьном виде.

Команды AAA и AAS преобразуют содержимое регистра AL в правильную неупакованную десятичную цифру. Если результат корректировки превышает 9, то эти команды добавляют (вычитают — AAS) к содержимому регистра AH и устанавливают флаги CF и OF. Данные корректирующие команды лучше сразу использовать после команды сложения (вычитания) чисел.

Пример корректировки сложения:

```
mov ax,5
mov bx,6
```

```
add ax,bx
aaa
```

В результате в регистре AX окажется значение 0000 0001 0000 0001 или 101h (можно увидеть под отладчиком).

Пример корректировки вычитания:

```
mov ax,5
mov bx,6
sub ax,bx
aas
```

5.2.5.2. Умножение и деление неупакованных BCD-чисел

Для корректировки умножения и деления неупакованных BCD-чисел имеются две команды:

- AAM (ASCII Adjust for Multiplication) — коррекция умножения для представления в символьном виде;
- AAD (ASCII Adjust for Division) — коррекция деления для представления в символьном виде.

Команды AAM и AAD преобразуют содержимое регистра AX в правильную неупакованную десятичную цифру.

Обратите внимание: в отличие от других корректирующих команд (AAA, DAA и др.), которые выполняют действия над результатом операции, команда AAD должна выполняться непосредственно *перед операцией* деления!

Команды AAM и AAD устанавливают флаги SF, ZF и PF в соответствии с результатом, OF, AF и CF остаются неопределенными.

5.2.5.3. Сложение и вычитание упакованных BCD-чисел

Можно выполнять сложение и вычитание чисел в упакованном двоично-десятичном представлении (BCD-формате):

Для этих целей имеются две корректирующие команды:

- DAA (Decimal Adjustment for Addition — десятичная коррекция для сложения);
- DAS (Decimal Adjustment for Subtraction — десятичная коррекция для вычитания).

Команды DAA и DAS преобразуют содержимое регистра AL в правильную упакованную десятичную цифру. Если результат корректировки превышает предельное значение для упакованных BCD-чисел (99), то эти команды добавляют (вычитают — DAS) к содержимому регистра AH единицу и устанавливают флаги AF и CF.

Обработка полей также осуществляется по одному байту за одно выполнение.

5.3. Команды выполнения операций с вещественными числами

Число, которое имеет запятую для разделения целой и дробной части, называют вещественным (или действительным) числом.

Существует две формы представления вещественных чисел:

- с фиксированной запятой (естественная форма);
- с плавающей запятой (экспоненциальная форма).

Вещественное число в естественной форме имеет фиксированное место запятой — если место запятой изменить, то изменится и само число.

Примеры чисел с фиксированной запятой:

```
a = 0.3702
b = -324.8451
c = 2.7183
```

d = 572.0

Например, числа 45.13 и 4.513 — это совершенно разные числа, несмотря на одинаковый порядок цифр. Таким образом, положение запятой в вещественных числах с фиксированной запятой имеет принципиальное значение.

В вещественных числах с плавающей запятой текущее положение запятой в числе не принципиально, т. к. реальное место запятой определяется с помощью показателя степени.

Примеры чисел с плавающей запятой:

a = $-0.248451 \cdot 10^{2}$

b = $3.56 \cdot 10^{-3}$

c = $0,1082 \cdot 10^{4}$

d = $58.054 \cdot 10^{27}$

Например, числа $1.3 \cdot 10^{1}$, $0.13 \cdot 10^{2}$, $0.013 \cdot 10^{3}$ и $0.0013 \cdot 10^{4}$ — это одно и тоже число 13, хотя запятая у всех этих чисел находится на разной позиции – именно поэтому такие вещественные числа называют "с плавающей запятой".

Вещественные числа с плавающей запятой используют в основном для компактной записи очень больших чисел наподобие числа Авогадро $N=6.02214 \times 10^{23}$ или очень малых чисел, как постоянная планка $h=6.6261 \times 10^{-27}$ эрг•с.

Именно для удобной записи и оперирования очень большими и очень малыми числами была изобретена форма записи чисел с плавающей запятой.

В компьютере можно работать как с вещественными числами с фиксированной запятой, так и с числами с плавающей запятой.

На ассемблере работа с числами с фиксированной запятой осуществляется почти точно также как с целыми числами. Для оперирования числами с плавающей запятой можно написать программную реализацию или воспользоваться встроенной поддержкой — сопроцессором.

Далее мы подробно рассмотрим работу с вещественными числами с фиксированной запятой и отдельно с плавающей запятой.

5.3.1. Вычисления с фиксированной запятой

Числа с фиксированной запятой, в отличие от чисел с плавающей запятой, имеют фиксированное количество разрядов, выделенных на целую (до запятой) и дробную (после запятой) части. Например, пусть в десятичной системе счисления выделено 5 разрядов для целой части числа (до запятой) и 5 разрядов для дробной части числа (после запятой); числа, записанные в такую разрядную сетку, имеют вид:

+00721,35500; +00000,00328; -10301,20260.

В компьютере наиболее распространенные форматы чисел с фиксированной запятой — 8:8 (всего 16 бит) и 16:16 (всего 32 бита). Разумеется, можно придумать и использовать иной формат, например 6:10, но это усложнит некоторые операции над такими числами.

Диапазон значащих чисел N в системе счисления с основанием P при наличии m разрядов в целой части и s разрядов в дробной части числа (без учета знака числа) будет:

$P^{-s} \le N \le P^{m}-1$.

Таким образом, для формата 8:8 (m=8, s=8) в двоичной системе счисления (P=2) диапазон будет:

$0,00390625 \le N \le 255$.

Для 16:16:

$0,0000152587890625 \le N \le 65535$.

При P=2, m=16 и s=0: $0 \le N \le 65535$.

При P=2, m=32 и s=0: $0 \le N \le 4294967295$.

Вещественные числа в компьютере хранятся также как и все данные в двоичном виде. Однако нужно помнить, что вещественные числа из недвоичной системы счисления переводятся в двоичную и обратно не так как обычные целые числа.

Причем для вещественных чисел с фиксированной и плавающей запятой такой перевод осуществляется по-разному.

На первом дне я говорил, что для перевода целых чисел из одной системы счисления в другую вы можете воспользоваться обычным калькулятором. Однако в случае вещественных чисел калькулятор будет плохим помощником. Например, попробуйте с помощью стандартного калькулятора Windows перевести вещественное число 43,58 в десятичной системе счисления в любую другую систему счисления – двоичную или шестнадцатеричную. Калькулятор переведет только целую часть, а дробную просто отбросит как ненужную. Обратный перевод, например из двоичного числа в вещественное число в десятичной системе счисления вообще с помощью калькулятора напрямую выполнить невозможно.

Поэтому мне ничего не остается, как немного рассказать вам как осуществляется перевод вещественных чисел с фиксированной запятой из десятичной системы счисления в двоичную и обратно.

Целая часть (до запятой) вещественного числа с фиксированной запятой переводится в двоичный вид также как любые целые числа. А для перевода дробной части (после запятой) используют следующий алгоритм. Число в десятичной системе счисления необходимо последовательно умножать на основание системы счисления, в которую надо переводить. Причем умножать надо только очередную дробную часть, игнорируя возникающие целые части. В качестве цифр берутся целые части результатов умножения. Так происходит до тех пор, пока дробная часть не станет равной нулю или не достигается заданная точность.

Ниже приводится пример перевода вещественного числа с фиксированной запятой 213,625 в двоичную систему счисления.

Переводим целую часть с помощью калькулятора или по правилу, рассказанному на первом дне: 213d = 11010101b.

Теперь переводим дробную часть:

```
0,625 умножаем на 2. Дробная часть 0,250. Целая часть 1.
0,250 умножаем на 2. Дробная часть 0,500. Целая часть 0.
0,500 умножаем на 2. Дробная часть 0,000. Целая часть 1.
```

Итак, сверху вниз получаем число 101.

Результат: 213,625d = 11010101,101b.

Обратный перевод из двоичного вида в десятичный вид вещественного числа с фиксированной запятой выполняется почти также как перевод целых чисел только в дробной части степени будут отрицательными. Пример:

$$00010000.10000000 = 0x2^7+0x2^6 + 0x2^5 + 1x2^4 + 0x2^3 + 0x2^2 + 0x2^1 + 0x2^0$$
$$+1x2^{-1} +0x2^{-2} +0x2^{-3} + 0x2^{-4} + 0x2^{-5} + 0x2^{-6} + 0x2^{-7} = 16 + 0,5 = 16,5d$$

Сложение и вычитание чисел с фиксированной запятой выполняется точно также как сложение и вычитание целых чисел:

```
mov ax,1080h
mov bx,1240h
add ax,bx
sub ax,bx
```

При выполнении умножения чисел с фиксированной запятой умножение 16-битных чисел дает 32-битный результат, а умножение 32-битных чисел – 64-битный результат. Например, если в регистрах EAX и EBX содержатся числа с фиксированной запятой в формате 16:16, то после выполнения умножения:

```
xor edx,edx
mul ebx
```

в регистр EDX будет помещена вся целая часть, а в регистр EAX — вся дробная.

Форма чисел с фиксированной запятой наиболее проста и естественна, но имеет небольшой диапазон представления чисел и поэтому не всегда приемлема при вычислениях.

5.3.2. Вычисления с плавающей запятой

Числа с плавающей точкой — это числа, представляемые в экспоненциальной форме, т. е. в таком виде:

$X = S \times M \times N^q$,

где X – вещественное число,

S — знак числа,

M — мантисса числа,

N — основание системы счисления,

q — экспонента называемая часто порядком (целое). Порядок (q) определяет положение запятой в мантиссе.

Например, в числе 6.6261×10^{-27}: M=6.6261, N=10, q=-27, знак числа "+" (S).

Данная форма позволяет перемещать десятичную запятую в вещественном числе вправо и влево, не меняя истинного значения числа.

Так как компьютер использует двоичную систему счисления, то N всегда равен 2.

Сопроцессор для преставления вещественных чисел с плавающей запятой руководствуется стандартом IEEE Standard for Binary Floating-Point Arithmetics, IEEE 754-1985.

Согласно этому стандарту старший разряд двоичного представления вещественного числа всегда кодирует знак числа: 0 – положительное, 1 – отрицательное. Остальная часть разбивается на две части: экспонента и мантисса.

Стандарт определяет три формата вещественных чисел с плавающей запятой (рис. 5.1):

- короткий (одинарной точности) — для хранения числа отводится 32 бита. Под мантиссу отводится 24 бита.
- длинный (двойной точности) — для хранения числа отводится 64 бита. Под мантиссу отводится 54 бита.
- расширенный (повышенной точности) — для хранения числа отводится 80 битов. Под мантиссу отводится 64 бита.

Рис. 5.1. Форматы чисел с плавающей запятой согласно стандарту IEEE 754-1985

Согласно стандарту IEEE 754-1985 порядок должен храниться в *смещенном формате*, который определяется по следующей формуле:

```
q = p + значение смещения
```

Значение q часто называют *характеристикой*.

Для коротко формата вещественных чисел значение смещения равно +127, для длинного +1023, для расширенного +16383.

Кроме того, согласно IEEE 754-1985 мантисса должна быть представлена в нормализованном виде:

```
M = 1.дробная часть
```

т. е. порядок (не характеристика) должен быть подобран такой величины, чтобы целая часть мантиссы была равна 1 (в двоичном представлении). Так как единица в целой части всегда присутствует, то сопроцессор отбрасывает целую часть мантиссы,

сохраняя лишь дробную часть. Это дает возможность увеличить диапазон представимых чисел, так как появляется лишний разряд, пригодный для представления мантиссы числа. Но это справедливо только для короткого и длинного форматов вещественных чисел. Расширенный формат как внутренний формат представления числа любого типа в сопроцессоре содержит целую единичную часть вещественного в явном виде.

Для определения чисел с плавающей запятой в программе можно использовать директивы MASM: real4 — для коротких, real8 — для длинных и real10 — для расширенных вещественных чисел. Пример:

```
Number1 real4 45.56 ; короткий формат
Number2 real8 45.56 ; длинный формат
Number3 real10 45.56 ; расширенный формат
```

Кроме этого, для определения чисел с плавающей запятой можно воспользоваться директивами DD — для коротких, DQ — для длинных и DT — для расширенных вещественных чисел. Однако директивы DD, DQ, DT не сообщают об ошибке, если число не содержит точки.

Примеры:

```
dd 45.56 ; короткий формат
dq 45.56 ; длинный формат
dt 45.56 ; расширенный формат
```

В данных директивах можно использовать символ "e" для экспоненциальной формы. Предыдущие определения с использованием "e" будут выглядеть следующим образом:

```
dd 0.4556e2    ; короткий формат
dq 0.4556e2    ; длинный формат
dt 0.4556e2    ; расширенный формат
```

Так как числа с плавающей запятой довольно сложно устроены, то к ним нельзя сразу применять обычные арифметические инструкции ассемблера, такие как ADD, SUB, MUL, DIV. Чтобы вести правильные арифметические расчеты с числами с плавающей запятой нужно каким-то образом выделять мантиссу и экспоненту, производить множество вспомогательных действий и результат снова упаковывать в 32, 64 или 80 бита.

На наше счастье сопроцессор берет обработку чисел с плавающей запятой на себя. Сопроцессор предоставляет особые инструкции, которые позволяют оперировать числами с плавающей запятой также просто как целыми числами, и предоставляет инструкции для реализации очень сложных алгоритмов.

Сопроцессор предоставляет программисту восемь регистров (R0-R7) для хранения данных и еще пять вспомогательных регистров (CWR,SWR,TWR,IPR,DPR) (рис. 1.9).

Команды сопроцессора не могут оперировать реальными названиями регистров хранения данных R0...R7, вместо них используются логические обозначения этих регистров ST(0)...ST(7).

В отличие от регистров центрального процессора регистры данных сопроцессора не работают независимо, а организованы в виде стека. Вершина стека всегда называется ST(0), затем идут ST(1), ST(2) и так далее до ST(7). Обычно вместо ST(0) используют краткое обозначение ST. Кстати, название ST происходит от английского слова STack (стек). Как и у ЦП, стек сопроцессора растет к регистрам с меньшими адресами.

Особенностью такого стека является еще то, что при помещении значения в него вершина стека ST смещается, а вместе с ней смещаются все логические номера ST(1)...ST(7) при этом смещение происходит циклически (по кольцу). Например, если текущая вершина стека ST соответствует регистру R3, то после записи в этот стек значения оно будет записано в регистр R2 и станет называться ST, при этом соответствующим образом сместятся все логические номера, а элемент соответствующий R0 станет соответствовать R7. Смещаются циклические только логические номера ST(1)...ST(7), а не сами значения в стеке!

В стеке сопроцессора могут храниться только 8 чисел, поэтому при добавлении в него 9 числа самое дальнее отстоящее от вершины число будет потеряно.

Из-за стековой структуры регистров сопроцессора форма записи математических выражений для FPU является обратной польской нотацией. Согласно этой нотации все операторы математического выражения указываются после своих операндов, например выражение a+b в польской нотации будет выглядеть как ab+, а выражение cos(x) как x cos. При этом отпадает необходимость использовать скобки, например: выражение (A+B)*(C+D)-E будет записано как AB+CD+*E-.

Все команды сопроцессора начинаются с буквы F, что является их отличительной чертой. Все основные команды сопроцессора перечислены в разд. 5.4.4.

Сопроцессор работает параллельно с центральным процессором, поэтому раньше (до 287) необходимо было для синхронизации работы двух процессоров перед многими командами FPU вставлять команду FWAIT (или WAIT), чтобы приостановить выполнение ЦП до окончания вычислений сопроцессором. В последующих сопроцессорах эта команда встроена в инструкции FPU, поэтому ее использовать не обязательно. Но остались не ожидающие команды, которые можно узнать по приставке "N". Например, команды FSAVE и FNSAVE выполняют одно и тоже действие, но перед FSAVE компилятор всегда добавляет команду FWAIT.

Если в команде присутствует суффикс "P", то это означает, что команда после своего выполнения выталкивает число из стека сопроцессора (помечает ST(0) как пустой и увеличивает TOP на один). К таким командам, например, относятся FADDP, FSUBP, FMULP, FDIVP, FCOMP.

Многие команды работают только с ST(0), но при необходимости можно легко использовать число из другого регистра данных сопроцессора, предварительно поменяв значения регистров командой FXCH.

Перед началом использования сопроцессора в программе необходимо выполнить его инициализацию командой FINIT. Эта команда устанавливает значения по умолчанию во всех регистрах и флагах FPU, в том числе обнуляет регистры данных ST(0)-ST(7).

В листинге 5.2 показан пример сложения двух вещественных чисел с плавающей запятой. Программа не выводит результат на экран, но по аналогии с программой из листинга 5.1 вы можете написать процедуры для ввода и вывода вещественных чисел (это будет вашим домашним заданием).

Ассемблирование и линковка выполняется обычным образом:

```
ml fpuadd.asm
```

Листинг 5.2. Сложение двух вещественных чисел (fpuadd.asm)

```
        .model tiny
        .code
        org     100h
start:
        finit           ; инициализировать сопроцессор
        fld     num1    ; загружаем 1-й операнд в ST
        fld     num2    ; 2-й операнд в ST, 1-й операнд в ST(1)
        fadd            ; складываем, сумма помещается в ST
        fstp    result  ; скопировать из ST в result

        ret

num1    dd      45.56   ; первое вещественное число
num2    dd      30.13   ; второе вещественное число
result  dt      ?       ; сюда будет помещен результат

        end     start
```

В листинге 5.3 показан более сложный пример — вычисление арктангенса по следующей формуле:

$$y = 2 * arctg(\frac{x^2}{\sqrt{x^2+1}})$$

Где -1 ≤ x ≤ +1.

В обратной польской нотации это уравнение будет выглядеть следующим образом:

```
x x * x x * 1 + sqrt / arctg 2 *
```

Обратите внимание, что в программе мы не выполняем явно деление (для чего обычно используется инструкция FDIV), т. к. это деление автоматически выполняет сама функция FPATAN.

Листинг 5.3. Вычисления арктангенса по формуле: y = 2*arctg(x²/sqrt(x²+1)) (fpuarctg.asm)

```
        .model tiny
        .code
        org     100h

start:
        finit           ; инициализировать сопроцессор
        fld     x       ; x в стек
        fld     x       ; x в стек еще раз
        fmul            ; перемножить (соответствует x^2)
        fld     st(0)   ; сделать копию st(0)
        fld1            ; поместить в стек 1,0
        fadd            ; выполнить операцию x^2+1
        fsqrt           ; взять корень sqrt(x^2+1)
        fpatan          ; арктангенс arctg(x^2/sqrt(x^2+1))
        fild    y       ; число 2 в стек
        fmul            ; перемножить 2*arctg(x^2/sqrt(x^2+1))

        ret

x       dd      0.5     ; значение x
y       dd      2       ; значение y

        end start
```

5.3.2.1. Сравнение вещественных чисел

Если вам понадобится в программе сравнить два вещественных числа, то у вас не получится использовать обычные инструкции сравнения, такие как TEST и CMP, т. к. они, во-первых, предназначены для сравнения целых чисел, а, во-вторых, не работают со стеком FPU. В сопроцессоре предусмотрены специальные команды сравнения: FTST, FCOM, FUCOM, FICOM, FCOMI и др. (см. их описание в разд. 5.4.4). Эти команды сравнения сохраняют результат сравнения в трех битах (флагах) C0, C2 и C3 специального слова состояния сопроцессора (регистр состояния SWR, см. разд. 5.4.2.2).

С помощью команды FSTSW AX можно сохранить значение регистра SWR в регистр AX, а затем командой SAHF перевести флаги C0, C2 и C3 в ZF, PF и CF. Это позволяет использовать условные команды Jcc, CMOVcc, FCMOVcc и др., точно также как после команды CMP.

Например, в следующем фрагменте программы выполняется переход к метке error, если операнды в ST(0) и ST(1) несравнимы:

```
fcom
```

```
fstsw   ax
sahf
je      error
```

5.4. Архитектура сопроцессора

5.4.1. Типы данных FPU

Сопроцессор может выполнять операции с семью различными типами данных, представленными в табл. 5.1.

Таблица 5.1. *Типы данных FPU*

Тип данных	Бит	Пределы
Целое число	16	-32768 ... 32767
Короткое целое	32	$-2x10^9$ ... $2x10^9$
Длинное слово	64	$-9x10^{18}$ ... $9x10^{18}$
Упакованное десятичное (BCD)	80	-99..99 ... +99..99 (18 цифр)
Короткое вещественное	32	$1{,}18x10^{-38}$ ... $3{,}40x10^{38}$
Длинное вещественное	64	$2{,}23x10^{-308}$ ... $1{,}79x10^{308}$
Расширенное вещественное	80	$3{,}37x10^{-4932}$ ... $1{,}18x10^{4932}$

Кроме обычных чисел в результате операций сопроцессора могут возникнуть также специальные случаи, над которыми можно также выполнять отдельные операции:

- положительный ноль (все биты числа нули);
- отрицательный ноль (знаковый бит равен 1, все остальные - нули);
- положительная бесконечность (знаковый бит 0, все биты мантиссы 0, все биты экспоненты 1);
- отрицательная бесконечность (знаковый бит 1, все биты мантиссы 0, все биты экспоненты 1);
- денормализованное число (все биты экспоненты 0);
- неопределенность (знаковый бит 1, все биты экспоненты 1, первый бит мантиссы (первые два для 80-битных чисел) – 1, остальные 0);
- не-число типа SNAN (сигнальное) (все биты экспоненты 1, первый бит мантиссы 0 (для 80-битных чисел первые два бита мантиссы – 10), а среди остальных есть единицы);
- не-число типа QNAN (тихое) (все биты экспоненты 1, первый бит мантиссы (первые два для 80-битных чисел) – 1, среди остальных есть единицы). Неопределенность – один из вариантов QNAN;
- неподдерживаемое число (ситуации, не соответствующие стандартным числам и не описанные в специальных случаях).

5.4.2. Регистры FPU

Регистры FPU показаны на рис. 1.9, ниже идет их описание.

5.4.2.1. Регистры данных R0-R7

К регистрам данных R0-R7 нельзя обращаться напрямую по их именам, поэтому в командах используются логические обозначения этих регистров ST(0)-ST(7). В отличие от регистров центрального процессора регистры данных сопроцессора не работают независимо, а организованы в виде стека. Вершина стека всегда называется ST(0), затем идут ST(1), ST(2) и так далее до ST(7). Вместо ST(0) обычно используется краткое обозначение ST. Подробнее см. разд. 5.3.2.

5.4.2.2. Регистр состояния SWR (Status Word Register)

Слово состояния отражает общее состояние сопроцессора:

- 0-й бит, флаг недопустимой операции (IE);
- 1-й бит, флаг денормализованной операции (DE) – выполнена операция над денормализованным числом;
- 2-й бит, флаг деления на ноль (ZE);
- 3-й бит, флаг переполнения (OE) – результат слишком большой;
- 4-й бит, флаг антипереполнения (UE) – результат слишком маленький;
- 5-й бит, флаг неточного результата (PE) – результат не может быть представлен точно;
- 6-й бит, ошибка стека (SF). Если C1=1, произошло переполнение (команда пыталась писать в непустую позицию в стеке), если C1=0, произошло антипереполнение (команда пыталась считать число из пустой позиции в стеке);
- 7-й бит, общий флаг ошибки (ES) – равен 1, если произошло хотя бы одно немаскированное исключение;
- 8-й бит, условный флаг 0 (C0);
- 9-й бит, условный флаг 1 (C1);
- 10-й бит, условный флаг 2 (C2);
- 11—13-й биты, число от 0 до 7, показывающее, какой из регистров данных R0—R7 является вершиной стека (TOP);
- 14-й бит, условный флаг 3 (C3);
- 15-й бит, флаг занятости FPU – этот флаг существует для совместимости с 8087, и его значение всегда совпадает с ES.

5.4.2.3. Регистр управления CWR (Control Word Register)

Слово управления сопроцессора определяет один из нескольких вариантов обработки численных данных:

- 0-й бит, маска недействительной операции (IM);
- 1-й бит, маска денормализованного операнда (DM);
- 2-й бит, маска деления на ноль (ZM);
- 3-й бит, маска переполнения (OM);
- 4-й бит, маска антипереполнения (UM);
- 5-й бит, маска неточного результата (PM);
- 6-й, 7-й биты, зарезервированы;
- 8-й, 9-й биты, управление точностью (PC) (см. табл. 5.2);
- 10-й, 11-й биты, управление округлением (RC) (см. табл. 5.3);
- 12-й, управление бесконечностью (IC) – поддерживается для совместимости с 8087 и 80287 – вне зависимости от значения этого бита $+\infty > -\infty$;
- 13—15-й биты, зарезервированы.

Биты PC определяют точность вычислений в сопроцессоре (табл. 5.2).

Таблица 5.2. *Точность результатов*

Значение PC	**Точность результатов**
0	Одинарная точность (32-битные числа)
01b	Зарезервировано
10b	Двойная точность (64-битные числа)
11b	Расширенная точность (80-битные числа) (этот режим устанавливается при инициализации сопроцессора)

Биты RC определяют способ округления результатов команд FPU до заданной точности (табл. 5.3).

Таблица 5.3. *Способы округления*

Значение RC	**Способ округления**
00b	К ближайшему числу (этот режим устанавливается при инициализации сопроцессора)
01b	К отрицательной бесконечности
10b	К положительной бесконечности
11b	К нулю

Биты 0-5 регистра CWR маскируют соответствующие исключения – если маскирующий бит установлен, исключения не происходит, а результат вызвавшей его команды определяется правилами для каждого исключения специально.

После выполнения команды FINIT в регистре управления устанавливается режим работы с расширенной точностью и округления к ближайшему числу. Все биты масок обработки исключений устанавливаются в 1; следовательно, все исключения будут замаскированы.

5.4.2.4. Регистр тегов TWR (Tags Word Register)

Регистр тегов сопроцессора содержит восемь пар битов (тегов), каждая из которых соответствует определенному физическому регистру стека, например: биты 15-14 описывают регистр R7, 13-12 – регистр R6 и т. д. Если содержимое одного из регистров R0-R7 изменяется, то это немедленно отражается на соответствующем этому регистру паре битов регистра тегов.

Значения, которые могут принимать пары битов (теги) в регистре тегов:

- 00 — регистр стека сопроцессора содержит допустимое ненулевое значение;
- 01 — регистр стека сопроцессора содержит нулевое значение;
- 10 — регистр стека сопроцессора содержит одно из специальных значений (кроме нуля): не-число, бесконечность, денормализованное число, неподдерживаемое число;
- 11 — регистр стека пуст.

5.4.2.5. Регистры-указатели команд IPR (Instruction Point Register) и данных DPR (Data Point Register)

Данные регистры используются в обработчиках исключений для анализа вызывавшей его команды. Регистр IRP содержит адрес последней выполненной команды, а DRP – адрес ее операнда.

5.4.3. Исключения FPU

При вычислениях с помощью команд сопроцессора могут возникать шесть типов особых ситуаций, называемых исключениями. Биты особых ситуаций хранятся в регистре состояния. Ниже перечислены все возможные исключения и действия выполняемые при наступившем исключении если соответствующий бит маски в регистре состояния установлен в 1:

- неточный результат (округление) — результат округляется в соответствии с битами RC. При этом флаг C1 показывает, в какую сторону произошло округление: 0 – вниз, 1 – вверх;
- недействительная операция — результат определяется из табл. 5.4;
- деление на ноль — результат преобразуется в бесконечность соответствующего знака (учитывается и знак нуля);
- антипереполнение (слишком маленький результат) — результат слишком мал, чтобы быть представленным обычным числом, — он преобразуется в денормализованное число;

- переполнение (слишком большой результат) — результат преобразуется в бесконечность соответствующего знака;
- денормализованный операнд — вычисление продолжается, как обычно.

Таблица 5.4. *Результаты операций, приводящих к исключениям*

Операция	**Результат**
Ошибка стека	Неопределенность
Операция с поддерживаемым числом	Неопределенность
Операция с SNAN	QNAN
Сравнение числа с NAN	C0=C2=C3=1
Сложение бесконечностей с одним знаком или вычитание – с разным	Неопределенность
Умножение нуля на бесконечность	Неопределенность
Деление бесконечности на бесконечность или 0/0	Неопределенность
Команды FPREM и FPREM1, если делитель – 0 или делимое – бесконечность	Неопределенность и C2=0
Тригонометрическая операция над бесконечностью	Неопределенность и C2=0
Корень или логарифм, если x<0, log(x+1), если x<-1	Неопределенность
FBSTP, если регистр-источник пуст, содержит NAN, бесконечность или превышает 18 десятичных знаков	Десятичная неопределенность
FXCH, если один из операндов пуст	Неопределенность

5.4.4. Команды сопроцессора

Команды сопроцессора можно сгруппировать в 6 функциональных классов:

- команды пересылки данных;
- арифметические команды;
- команды сравнения;
- трансцендентные команды;
- команды манипуляций константами;
- команды управления сопроцессором.

5.4.4.1. Команды пересылки данных FPU

Команды пересылки данных FPU показаны в табл. 5.5.

Таблица 5.5. *Команды пересылки данных FPU*

Команда	**Описание**
FLD источник	Загрузить вещественное число в ST(0) из источника (32-, 64- или 80-битная переменная или ST(n)). При этом TOP уменьшается на 1. Команда FLD ST(0) делает копию вершины стека
FILD источник	Загрузить целое число в ST(0) из источника (32-, 64- или 80-битная переменная). При этом целое число преобразовывается в вещественный формат и уменьшается TOP на 1
FBLD источник	Загрузить BCD-число в ST(0) из источника (80-битная переменная в памяти). При этом TOP уменьшается на 1
FST приемник	Скопировать вещественное число из ST(0) в приемник (32- или 64-битную переменную или пустой ST(n))
FSTP приемник	Переместить вещественное число из ST(0) в приемник (32-, 64- или 80-битную переменную или пустой ST(n)). При этом число выталкивается из стека, т. е. ST(0) помечается как пустое, а TOP увеличивается на один

Таблица 5.5. *(окончание)*

FIST приемник	Скопировать целое число из ST(0) в приемник (16- или 32-битную переменную)
FISTP приемник	Переместить целое число из ST(0) в приемник (16-, 32- или 64-битная переменная). При этом число выталкивается из стека, т. е. ST(0) помечается как пустое, а TOP увеличивается на один
FBST приемник	Скопировать BCD-число из ST(0) в приемник (80-битная переменная)
FBSTP приемник	Переместить BCD-число из ST(0) в приемник (80-битная переменная). При этом число выталкивается из стека, т. е. ST(0) помечается как пустое, а TOP увеличивается на один
FXCH ST(n)	Обменять местами содержимое двух регистров стека: ST(0) и ST(n). Если операнд не указан, то обмениваются ST(0) и ST(1)
FCMOVcc приемник, источник	Команда условной пересылки данных. Копирование содержимого источника (регистр ST(n)) в приемник (только ST(0)) осуществляется только если выполняется необходимое условие. Команда может принимать следующие формы: • FCMOVE — копировать, если равно (ZF=1); • FCMOVNE — копировать, если не равно (ZF=0); • FCMOVB — копировать, если меньше (CF=1); • FCMOVBE — копировать, если меньше или равно (CF=1 и ZF=1); • FCMOVNB — копировать, если меньше (CF=0); • FCMOVNBE — копировать, если меньше или равно (CF=0 и ZF=1); • FCMOVU — копировать, если не сравнимы (PF=1); • FCMOVNU — копировать, если сравнимы (PF=0)

5.4.4.2. Арифметические команды

Арифметические команды FPU приведены в табл. 5.6.

Таблица 5.6. *Арифметические команды FPU*

Команда	**Описание**
FADD приемник, источник	Сложение вещественных чисел. Команда складывает источник и приемник и помещает результат в приемник. Команда имеет следующие формы: FADD источник — источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником ST(0); FADD ST(0),ST(n), FADD ST(n),ST(0) — источником и приемником являются регистры FPU; FADD — без операндов эквивалентна FADD ST(0),ST(1)
FADDP ST(n), ST	Сложение с выталкиванием из стека. Команда складывает источник и приемник, помещает результат в приемник, после чего выталкивает ST(0) из стека (помечает ST(0) как пустой и увеличивает TOP на один). FADDP без операндов эквивалентна FADDP ST(1),ST(0)
FIADD источник	Сложение целых чисел. Источником является 16- или 32-битное целое число, а приемником – ST(0)
FSUB приемник, источник	Вычитание вещественных чисел. Команда вычитает источник из приемника и сохраняет результат в приемнике.Команда имеет следующие формы: FSUB источник — источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником ST(0); FSUB ST(0),ST(n), FSUB ST(n),ST(0) — источником и приемником являются регистры FPU; FSUB — без операндов эквивалентна FSUB ST(0),ST(1)

Таблица 5.6. *(продолжение)*

FSUBP ST(n), ST	Вычитание с выталкиванием из стека. Команда вычитает источник из приемника и сохраняет результат в приемнике, затем выталкивает ST(0) из стека и увеличивает TOP на один. FSUBP — без операндов эквивалентна FSUBP ST(1),ST(0)
FISUB источник	Сложение целых чисел. Источником является 16- или 32-битная переменная, содержащая целое число, а приемником ST(0)
FSUBR приемник, источник	Обратное вычитание вещественных чисел. Эта команда эквивалентна FSUB, но при этом выполняет вычитание приемника из источника, а не источника из приемника
FSUBRP приемник, источник	Обратное вычитание с выталкиванием. Эта команда эквивалентна FSUBRP, но при этом выполняет вычитание приемника из источника, а не источника из приемника
FISUBR источник	Обратное вычитание целых чисел. Эта команда эквивалентна FISUB, но при этом источником является ST(0), а приемником – 16- или 32-битная переменная, а не наоборот как в FISUB
FMUL приемник, источник	Умножение вещественных чисел. Команда выполняет умножение источника и приемника и помещает результат в приемник. Команда имеет следующие формы: FMUL источник — источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником ST(0); FMUL ST(0),ST(n), FMUL ST(n),ST(0) — источником и приемником являются регистры FPU; FMUL — без операндов эквивалентна FMUL ST(0),ST(1)
FMULP ST(n), ST	Умножение с выталкиванием из стека. Команда выполняет умножение источника и приемника и помещает результат в приемник, затем выталкивает ST(0) из стека и увеличивает TOP на один. FMULP — без операндов эквивалентна FMULP ST(1),ST(0)
FIMUL источник	Умножение целых чисел. Источником является 16- или 32-битная переменная, содержащая целое число, а приемником ST(0)
FDIV приемник, источник	Деление вещественных чисел. Команда выполняет деление приемника на источник и сохраняет результат в приемнике Команда имеет следующие формы: FDIV источник — источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником ST(0); FDIV ST(0),ST(n), FDIV ST(n),ST(0) — источником и приемником являются регистры FPU; FDIV — без операндов эквивалентна FDIV ST(0),ST(1)
FDIVP ST(n), ST	Деление с выталкиванием из стека. Команда выполняет деление приемника на источник и сохраняет результат в приемнике, затем выталкивает ST(0) из стека и увеличивает TOP на один. FDIVP — без операндов эквивалентна FDIVP ST(1),ST(0)
FIDIV источник	Деление целых чисел. Источником является 16- или 32-битная переменная, содержащая целое число, а приемником ST(0)
FDIVR приемник, источник	Обратное деление вещественных чисел. Эта команда эквивалентна FDIR, но при этом выполняет деление источника на приемник, а не приемника на источник
FDIVRP приемник, источник	Обратное деление с выталкиванием. Эта команда эквивалентна FDIVP, но при этом выполняет деление источника на приемник, а не приемника на источник
FIDIVR источник	Обратное деление целых чисел. Источником является ST(0), а приемником 16- или 32-битная переменная, содержащая целое число
FPREM	Нахождение остатка от деления. Выполняет деление ST(0) на ST(1) и помещает остаток от деления в ST(0)

Таблица 5.6. *(окончание)*

FPREM1	Нахождение остатка от деления в стандарте IEEE.
FABS	Нахождение абсолютного значения. Если в ST(0) отрицательное число, команда переводит его в положительное
FCHS	Изменение знака. Изменяет знак числа в ST(0) на противоположный
FRNDINT	Округление до ближайшего целого числа, находящегося в ST(0) в соответствии с режимом округления, заданным битами RC
FSCALE	Масштабирование. Умножает ST(0) на два в степени ST(1) и записывает результат в ST(0). Значение ST(1) предварительно округляется в сторону нуля до целого числа. Эта команда выполняет действие, обратное FXTRACT
FXTRACT	Извлечь экспоненту и мантиссу. Разделяет число в ST(0) на мантиссу и экспоненту, сохраняет экспоненту в ST(0) и помещает мантиссу в стек, так что после этого TOP уменьшается на 1, мантисса оказывается в ST(0), а экспонента – в ST(1)
FSQRT	Извлечь квадратный корень из ST(0) и поместить результат обратно в ST(0)

5.4.4.3. Команды манипуляций константами

Данные команды (табл. 5.7) помещают в ST(0) часто используемую константу.

Таблица 5.7. *Команды манипуляций константами FPU*

Команда	**Описание**
FLD1	Поместить в стек 1
FLDZ	Поместить в стек 0
FLDPI	Поместить в стек число π
FLDL2E	Поместить в стек $\log_2(e)$
FLDL2T	Поместить в стек $\log_2(10)$
FLDLN2	Поместить в стек ln(2)
FLDLG2	Поместить в стек lg(2)

5.4.4.4. Команды управления сопроцессором

Команды управления сопроцессором приведены в табл. 5.8.

Таблица 5.8. *Команды управления сопроцессором*

Команда	**Описание**
FINIT	Инициализация сопроцессора. Восстанавливает значения по умолчанию в регистрах CWR,SWR,TWR, а начиная с 80387 – IPR и DPR. Данная команда проверяет наличие произошедших и необработанных исключений и обрабатывает их до инициализации. Команда FINIT полностью эквивалентна команде WAIT FNINIT
FNINIT	Инициализация сопроцессора без ожидания. Восстанавливает значения по умолчанию в регистрах CWR,SWR,TWR, а начиная с 80387 – IPR и DPR
FSTSW приемник	Запись слова состояния (значение регистра SWR) в приемник (регистр AX или 16-битная переменная). Команда FSTSW AX обычно используется после команд сравнения
FNSTSW приемник	Запись слова состояния (значение регистра SWR) в приемник (регистр AX или 16-битная переменная) без ожидания
FLDCW источник	Загрузить регистр CWR. Копирует содержимое источника (16-битная переменная) в регистр CWR

Таблица 5.8. *(окончание)*

FCLEX	Обнулить флаги исключений (PE, UE, OE, ZE, DE, IE, а также флаги ES, SF и B в регистре состояния FPU). Данная команда проверяет наличие произошедших и необработанных исключений и обрабатывает их до выполнения. Команда FCLEX полностью эквивалентна команде WAIT FNCLEX
FNCLEX	Обнулить флаги исключений (PE, UE, OE, ZE, DE, IE, а также флаги ES, SF и B в регистре состояния FPU) без ожидания
FSTENV приемник	Сохранить состояние сопроцессора (CWR, SWR, TWR, IPR, DPR) в приемник (14 или 28 байт в памяти, в зависимости от разрядности операндов). Данная команда проверяет наличие произошедших и необработанных исключений и обрабатывает их до выполнения. Команда FSTENV полностью эквивалентна WAIT FNSTENV
FNSTENV приемник	Сохранить состояние сопроцессора (CWR, SWR, TWR, IPR, DPR) в приемник (14 или 28 байт в памяти, в зависимости от разрядности операндов) без ожидания
FLDENV источник	Загрузить состояние сопроцессора (CWR, SWR, TWR, IPR, DPR) из источника (область памяти в 14 или 28 байт, в зависимости от разрядности операндов)
FSAVE приемник	Сохранить состояние FPU. Сохраняет регистры данных и вспомогательные регистры в приемник (область памяти 94 или 108 байт, в зависимости от разрядности операндов) и инициализирует FPU аналогично командам FINIT/FNINIT. Данная команда проверяет наличие произошедших и необработанных исключений и обрабатывает их до выполнения. Команда FSAVE полностью эквивалентна WAIT FNSAVE
FNSAVE приемник	Сохранить состояние FPU без ожидания. Сохраняет регистры данных и вспомогательные регистры в приемник (область памяти 94 или 108 байт, в зависимости от разрядности операндов) и инициализирует FPU аналогично командам FINIT/FNINIT
FXSAVE приемник	Быстрое сохранение состояния FPU. Данная команда работает только на процессорах Pentium II и выше. Сохраняет регистры данных и вспомогательные регистры в приемник (512-байтную область памяти с адресом, кратным 16), не проверяя на необработанные исключения, аналогично команде FNSAVE
FRSTOR источник	Восстановить состояние FPU. Загружает регистры данных и вспомогательные регистры из источника (область в памяти размером в 94 или 108 байт, в зависимости от разрядности операндов)
FXRSTOR источник	Быстрое восстановление состояния FPU. Данная команда работает только на процессорах Pentium II и выше. Восстанавливает текущее состояние FPU, включая все регистры данных и вспомогательные регистры из источника (512-байтной области памяти с адресом, кратным 16), который был запомнен командой FXSAVE
FINCSTP	Инкремент указателя стека. Увеличивает на 1 значение поля TOP регистра состояния FPU. Если TOP было равно семи, оно обнуляется
FDECSTP	Декремент указателя стека. Уменьшает на 1 значение поля TOP регистра состояния FPU. Если TOP было равно нулю, оно устанавливается в 7
FFREE ST(n)	Освободить регистр данных. Команда отмечает в регистре TWR, что указанный регистр данных ST(n) пустой. Содержимое регистра TOP не изменяется
FNOP	Холостая операция сопроцессора. Данная команда занимает место и время, но не выполняет никакого действия
WAIT (FWAIT)	Ожидание процессором завершения текущей операции сопроцессора. WAIT и FWAIT – это разные названия одной и той же команды

5.4.4.5. Команды сравнения

Команды сравнения FPU приведены в табл. 5.9.

Таблица 5.9. *Команды сравнения FPU*

Команда	**Описание**
FCOM источник	Сравнение вещественных чисел хранящихся в ST(0) и в источнике. Источник может быть 32- или 64-битная переменная или регистр ST(n). Если источник не указан, сравниваются ST(0) и ST(1). При этом устанавливаются флаги C0, C2 и C3 следующим образом: • ST(0) > источник C0=0, C2=0, C3=0 • ST(0) < источник C0=1, C2=0, C3=0 • ST(0) = источник C0=0, C2=0, C3=1 Если операнды несравнимы, то C0=C2=C3=1
FCOMP источник	Сравнить и вытолкнуть из стека. Сравнивает вещественного число в ST(0) с источником, затем выталкивает содержимое ST(0) из стека и увеличивает TOP на 1. Источник может быть 32- или 64-битная переменная или регистр ST(n). Флаги C0, C2 и C3 устанавливаются также как в команде FCOM
FCOMPP	Сравнивает ST(0) и ST(1) и выталкивает из стека и ST(0), и ST(1). Флаги C0, C2 и C3 устанавливаются также как в команде FCOM
FUCOM ST(n)	Сравнение ST(0) с ST(n) без учета порядков. Флаги C0, C2 и C3 устанавливаются также как в команде FCOM
FUCOMP ST(n)	Сравнение ST(0) с ST(n) без учета порядков. При выполнении операции происходит выталкивание из стека ST(0). Флаги C0, C2 и C3 устанавливаются также как в команде FCOM
FUCOMPP ST(n)	Сравнение ST(0) с ST(n) без учета порядков. При выполнении операции происходит выталкивание из стека и ST(0), и ST(n). Флаги C0, C2 и C3 устанавливаются также как в команде FCOM
FICOM источник	Сравнение целых чисел в ST(0) с источником (16-, 32-битная переменная или ST(n))
FICOMP источник	Сравнение целых чисел в ST(0) с источником (16-, 32-битная переменная или ST(n)). При выполнении операции происходит выталкивание ST(0) из стека
FCOMI источник	Сравнить и установить EFLAGS. Выполняет сравнение ST(0) и источника (регистр ST(n)) и устанавливает флаги следующим образом: • ST(0) > источник ZF=0, PF=0, CF=0; • ST(0) < источник ZF=0, PF=0, CF=1; • ST(0) = источник ZF=1, PF=0, CF=0. Если операнды несравнимы, то все три флага равны 1
FCOMIP источник	Сравнить, установить EFLAGS и вытолкнуть. Флаги устанавливаются также как в команде FCOMI
FUCOMI источник	Сравнить без учета порядков и установить EFLAGS. Флаги устанавливаются также как в команде FCOMI
FUCOMIP источник	Сравнить без учета порядков, установить EFLAGS и вытолкнуть из стека. Флаги устанавливаются также как в команде FCOMI
FTST	Проверка ST(0) на ноль. Сравнивает содержимое ST(0) с нулем и выставляет флаги C3, C2 и C0 аналогично другим командам сравнения
FXAM	Анализ содержимого вершины стека (ST(0)). Устанавливает флаги C3, C2, C0 следующим образом: • 000 — неподдерживаемый формат; • 001 — не число; • 010 — нормализованное число; • 011 — бесконечность; • 100 — ноль; • 101 — пустой операнд; • 110 — денормализованное число

5.4.4.6. Трансцендентные команды

Трансцедентные команды FPU приведены в табл. 5.10.

Таблица 5.10. *Трансцендентные команды FPU*

Команда	**Описание**
FSIN	Вычисляет синус значения находящегося в ST(0), и сохраняет результат в этом же регистре. Значение считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63}. Если значение выходит за эти пределы, флаг C2 устанавливается в 1 и содержимое ST(0) не изменяется
FCOS	Вычисляет косинус значения находящегося в ST(0), и сохраняет результат в этом же регистре. Значение считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63}. Если значение выходит за эти пределы, флаг C2 устанавливается в 1 и содержимое ST(0) не изменяется
FSINCOS	Вычисляет синус и косинус значения находящегося в ST(0), результат вычисления синуса оказывается в ST(1), а косинуса – в ST(0), TOP уменьшается на 1. Значение считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63}. Если значение выходит за эти пределы, флаг C2 устанавливается в 1 и содержимое ST(0) не изменяется
FPTAN	Вычисляет тангенс числа, находящегося в регистре ST(0) и сохраняет результат в этом же регистре, а затем в стек проталкивается 1, поэтому результат оказывается в ST(1), а ST(0) содержит 1, TOP уменьшается на единицу. Значение считается заданным в радианах и не может быть больше 2^{63} или меньше -2^{63}. Если значение выходит за эти пределы, флаг C2 устанавливается в 1 и содержимое ST(0) и стек не изменяются. Единица помещается в стек для того, чтобы можно было получить котангенс вызовом команды FDIVR сразу после FPTAN
FPATAN	Вычисляет арктангенс числа: arctg(ST(1)/ST(0)). После вычисления происходит выталкивание из стека, в итоге результат оказывается в вершине стека
F2XM1	Вычисление 2^x-1. Возводит 2 в степень, значение которой расположено в ST(0), и вычитает 1. Результат сохраняется в ST(0). Значение ST(0) должно лежать в пределах от -1 до +1, иначе результат не определен
FYL2X	Вычисление $y \times \log_2(x)$. Вычисляет $ST(1) \times \log_2(ST(0))$, помещает результат в ST(1) и выталкивает ST(0) из стека, так что после этой операции результат оказывается в ST(0). Первоначальное значение ST(0) должно быть неотрицательным
FYL2XP1	Вычисление $y \times \log_2(x+1)$. Вычисляет $ST(1) \times \log_2(ST(0)+1)$, помещает результат в ST(1) и выталкивает ST(0) из стека, так что после этой операции результат оказывается в ST(0). Первоначальное значение ST(0) должно быть в пределах от $-(1-\sqrt{2}/2)$ до $(1+\sqrt{2}/2)$, в противном случае результат не определен

ДЕНЬ 6

Программирование под MS-DOS

Почти все, что мы рассмотрели на предыдущих днях (архитектура компьютера, конструкции и команды ассемблера), касается непосредственно ассемблера, но не программирования под MS-DOS.

Конечно, в наше время программирование под MS-DOS уже неактуально. Однако, многие простые программы на ассемблере иногда быстрее и проще реализовать под MS-DOS, чем с использованием функций API под Windows. В любом случае я советую ознакомиться с материалом этого дня, т. к. некоторые сведения здесь позволяют лучше узнать архитектуру компьютера (скан-коды, аппаратные и программные прерывания, работа с видеопамятью и пр.).

Разумеется, за один день невозможно рассмотреть все особенности программирования под MS-DOS (да это и не нужно в наше время), поэтому здесь приводится только все самое необходимое без чего невозможно создание более менее полноценного приложения: чтение параметров командной строки, вывод на экран в текстовом режиме, ввод с клавиатуры, работа с файлами и пр.

6.1. Чтение параметров командной строки

На втором дне (см. разд. 2.8) мы говорили о том, что параметры командной строки, переданные программе при старте, сохраняются в структуре PSP по адресу PSP:80h.

Например, если файл был запущен следующим образом:

```
program.exe par1 par2 par3
```

то здесь "`program.exe par1 par2 par3`" — командная строка, а "` par1 par2 par3`" — параметры командной строки.

Первый байт, расположенный в PSP по смещению 80h, всегда указывает длину строки параметров (см. табл. 2.2), т. е. количество символов вместе с пробелами (для примера выше длина будет равен 15). Понятно, что если файлу никаких параметров передано не было, то его значение будет равно нулю. Второй байт является пробелом (20h) — он отделяет имя файла от первого параметра, после которого идут все параметры, разделяемые пробелами. За самым последним параметром всегда располагается код клавиши <Enter> (0Dh).

В листинге 6.1 показана программа, которая принимает параметры командной строки, затем считывает параметры из PSP и выводит на экран в том же виде, в каком они были указаны в командной строке.

Параметры командной строки можно наблюдать в отладчике (рис. 6.1) в окне 5 (содержимое памяти). Для этого запустите программу под отладчиком следующим образом:

```
cv psp_arg.com one two three param
```

где `one two three param` — параметры командной строки.

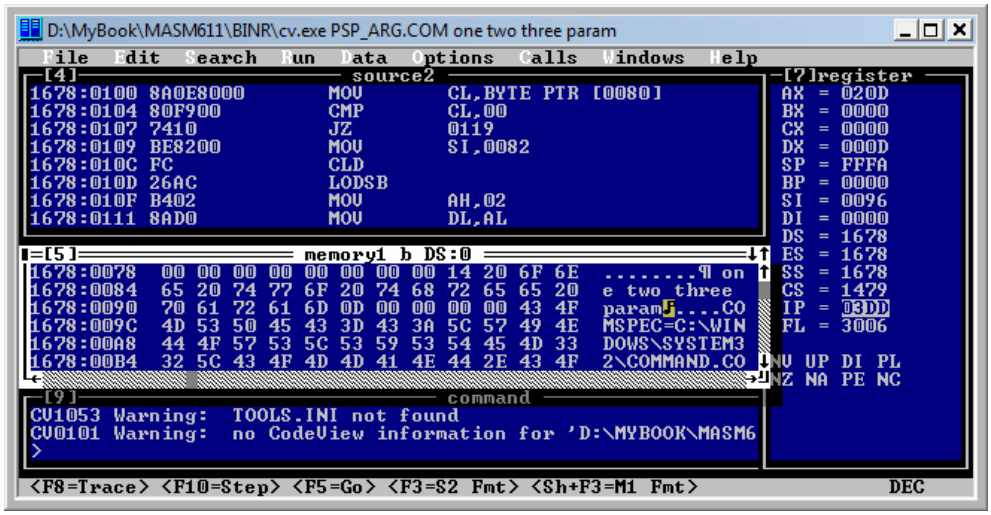

Рис. 6.1. Параметры командной строки в PSP после загрузки программы

Листинг 6.1. Пример считывания параметров командной строки из PSP (psp_arg.asm)

```
        .model tiny
        .code
        org     100h
start:
        mov     cl,ds:80h       ; количество символов
                                ; в командной строке

        cmp     cl,0            ; если нет параметров - выходим
        je      err_arg         ; с соответствующим сообщением

        mov     si,82h          ; по смещению 82h
                                ; размещается командная строка
       cld
get_arg:
        lods    byte ptr es:[si] ; помещаем в al очередной
                                 ; символ командной строки

        mov     ah,2             ; выводим символ на экран
        mov     dl,al
        int     21h

        loop    get_arg
        jmp     end_progr
err_arg:
        mov      ah,09h
        mov      dx,offset no_arg
        int      21h
end_progr:
        ret

no_arg  db      "Нет параметров",13,10,'$'

       end      start
```

6.2. Вывод на экран в текстовом режиме

6.2.1. Функции DOS

Мы уже неоднократно использовали функцию 09 (INT 21h) для вывода строк на экран (см. например листинги 2.1 и 2.2). В DOS существуют другие функций для вывода строк и отдельных символов на экран, ниже перечислены они все. В листинге 6.3 можно увидеть пример использования функции 02h (INT 21h) для вывода символов на экран.

02h (INT 21h) — вывод символа с проверкой на <Ctrl>+<Break>

Входные данные:

- AH=02h
- DL=ASCII-код символа

Возвращаемые значения:

- Ничего не возвращается.

Примечание. При использовании данной функции в случае ввода <Ctrl>+<C> или <Ctrl>+<Break>, вызывается прерывание 23h, которое по умолчанию осуществляет выход из программы.

06h (INT 21h) — вывод символа без проверки на <Ctrl>+<Break>

Входные данные:

- AH=06h
- DL=ASCII-код символа (кроме 0FFh)

Возвращаемые значения:

- Ничего не возвращается.

Примечание. Отличие от предыдущей функции только в том, что она не проверяет нажатие <Ctrl>+<C> или <Ctrl>+<Break>.

09h (INT 21h) — вывод строки на экран с проверкой на <Ctrl>+<Break>

Входные данные:

- AH=09h
- DS:DX=адрес строки, заканчивающейся символом $ (24h)

Возвращаемые значения:

- Ничего не возвращается.

40h (INT 21h) — записать в файл или на устройство

Входные данные:

- AH=40h
- BX=1 для вывода на STDOUT или BX=2 на устройство STDERR
- DS:DX=адрес начала строки
- CX=длина строки

Возвращаемые значения:

- CF=0
- AX=число записанных байтов

INT 29h — быстрый вывод символа на экран

Входные данные:

- AL=ASCII-код символа

Возвращаемые значения:

- Ничего не возвращается.

6.2.2. Прямая запись в видеопамять

Существует возможность выводить текст на экран без использования функций DOS и BIOS. Это достигается с помощью прямого обращения к видеопамяти компьютера.

Для текстовых режимов стандартно отводится область памяти, начинающаяся с абсолютного адреса 0B800h:0000h и заканчивающаяся 0B800h:0FFFFh. Любая информация, записанная в эту область, сразу пересылается в память видеоадаптера.

Под каждый символ отводится 2 байта: младший байт содержит ASCII-код отображаемого символа, старший байт — атрибут символа. Таким образом, каждый символ отстоит друг от друга на расстоянии 2 байта.

Атрибут состоит из двух частей: цвета фона символа и цвета самого символа. Коды атрибутов (цветов) приведены в табл. 6.1. Например, атрибут 034h означает, что цвет фона символа бирюзовый (3h), а цвет самого символа красный (4h).

Первые восемьдесят 2-байтовых полей соответствуют первой строке экрана, вторые 80 полей – второй строке и т. д. То есть переход на следующую строку экрана определяется не управляющими кодами возврата каретки и перевода строки, а размещением кодов символов в другом месте видеопамяти, в полях, соответствующих следующей строке. При прямой записи в видеопамять, в обход программ DOS и BIOS, все управляющие коды ASCII теряют свои управляющие функции и просто отображаются в виде соответствующих им символов.

В листинге 6.2 показана программа, которая выводит в центре экрана символ '+' прямым обращением к видеопамяти компьютера. Но таким способом неудобно выводить большие строки, поэтому в листинге 6.3 приведен другой вариант, который выводит разноцветное слово "Ivan" в верхнем левом углу с использованием инструкции `rep movsb`.

Таблица 6.1. *Коды цветов*

Код	Цвет	Код	Цвет
0h	Черный	8h	Серый
1h	Синий	9h	Голубой
2h	Зеленый	0Ah	Салатовый
3h	Бирюзовый	0Bh	Светло-бирюзовый
4h	Красный	0Ch	Розовый
5h	Фиолетовый	0Dh	Светло-фиолетовый
6h	Коричневый	0Eh	Желтый
7h	Белый	0Fh	Ярко-белый

Листинг 6.2. Вывод символа в центре экрана прямым обращением к видеопамяти (outchar1.asm)

```
        .model tiny
        .code
        org     100h
start:
        mov     ax,0003h        ; видеорежим 3 (очистка экрана)
        int     10h

        mov     ax,0B800h       ; сегментный адрес видеопамяти
        mov     es,ax
```

```
        mov     di,80*2*12+40*2 ; смещение начала в центр видеопамяти

        mov     ah,30           ; атрибут символа (цвет, фон)
        mov     al,'+'          ; код символа ("+")

        ; копируем символ и атрибут в видеопамять
        mov     word ptr es:[di],ax

        int     20h             ; выход из программы

        end     start
```

Листинг 6.3. Вывод слова в верхнем левом углу экрана прямым обращением к видеопамяти (outchar2.asm)

```
        .model tiny
        .code
        org     100h
start:
        mov     ax,0003h        ; видеорежим 3 (очистка экрана)
        int     10h
        cld                     ; обработка строк в прямом направлении

        mov     ax,0B800h       ; сегментный адрес видеопамяти
        mov     es,ax

        mov     di,0            ; адрес начала видеопамяти в ES:DI
        mov     si,offset txt   ; в SI начало строки txt
        mov     cx,txtlen       ; в CX длина строки txt
        cld

        rep     movsb           ; копирование символов строки в видеопамять

        int     20h             ; выход из программы

txt     db 'I',034h,'v',0E9h,'a',0C5h,'n',05Dh
txtlen = $-txt

        end     start
```

6.3. Ввод с клавиатуры

Прежде чем рассмотреть способы ввода символов с клавиатуры, вам необходимо познакомиться с некоторыми важными понятиями, которые часто встречаются в описаниях функций DOS и BIOS:

- Скан-коды (scan-code) — это встроенные уникальные номера, которые имеет каждая клавиша клавиатуры (см. табл. П2.9). Причем каждая клавиша генерирует два типа скан-кодов: при нажатии (make-код) и при отпускании клавиши (break-код). Скан-код нажатия содержит в старшем разряде 1, а скан-код отпускания 0; в оставшихся разрядах располагается код-идентификатор клавиши. Поэтому скан-коды нажатия отличаются от соответствующих скан-кодов отпускания на значение 128 (80h). К примеру, если скан-код отпускания равен 0000 0101 (5), то соответствующий скан-код нажатия 1000 0101 (133), разница между ними составляет 133-5=128. Только не путайте скан-коды с ASCII-кодами и прочими

кодировками символов. Скан-код — это, по сути, заводской идентификатор клавиши, который никогда не изменяется. Например, в английской раскладке на клавише со скан-кодом 11h изображена буква W, в русской раскладке этой же клавише соответствует буква Ц, во французской раскладке на этой клавише изображена буква Z, на японских и китайских клавиатурах изображены соответствующие иероглифы (на сайте Microsoft **http://www.microsoft.com/globaldev/reference/keyboards.mspx** можно увидеть практически все известные в мире национальные раскладки). Таким образом, скан-код однозначно идентифицирует клавишу, но не позволяет определить, какую букву (латинскую, русскую и пр.) в нижнем или верхнем регистре вводит пользователь. Длина генерируемого скан-кода клавиши может быть от 1 до 6 байт, который можно получить чтением порта ввода-вывода 0x60. Например, все "серые клавиши" имеют префикс 0E0h. Так, клавиша <Ins> при нажатии создаст пару скан-кодов: 0E0h, 52h.

- Расширенные ASCII-коды — это коды функциональных и различных комбинаций клавиш, которые не имеют представляющего их символа ASCII. Расширенные коды имеют длину 2 байта, причем первый байт всегда равен 0. Второй байт — номер расширенного кода, список которых приведен в табл. П2.8. Некоторые расширенные коды совпадают со скан-кодами соответствующих клавиш. Например, код 0h 41h является кодом клавиши <F7>. Начальный ноль позволяет программе определить, что данный код принадлежит расширенному набору ASCII.
- Эхо — под этим словом понимается "отображение символов на экране". Например, если в описании функции сказано, что она выполняет ввод символа с эхо, то это означает, что вводимый символ будет отображаться на экране. Без эха вводимый символ не будет видно на экране.

DOS предоставляет 7 функций для ввода данных с клавиатуры. Кроме этих семи функций DOS позволяет обращаться к клавиатуре, как к файлу, с помощью функции 3Fh прерывания INT 21h. В BIOS также имеет несколько функций, вызываемые по прерыванию INT 16h, для ввода данных с клавиатуры. Далее приведено описание всех этих функций.

6.3.1. Функции DOS

01h (INT 21h) — ввод символа с эхо

Функция DOS 01h прерывания INT 21h

Входные данные:

- AH=01h

Возвращаемые значения:

- AL = ASCII-код символа или 0. Если AL=0, повторный вызов этой функции возвратит в AL расширенный ASCII-код символа.

Примечание. Если считывается комбинация клавиш <Ctrl>+<C> или <Ctrl>+<Break>, то выполняется прерывание INT 23h.

Для чтения расширенных ASCII-кодов необходимо обращаться к данной функции дважды. При первом обращении в AL возвращается значение 0, что указывает на наличие расширенного кода.

06h (INT 21h) — ввод-вывод через консоль

Данная функция способна выполнять как ввод, так и вывод с консоли (без ожидания) — это зависит от передаваемых значениях в регистре DL при вызове.

Входные данные:

- AH=06h
- DL = запрошенная функция. Если DL=00h–0FEh, то запрашивается вывод. Если DL=0FFh.

Возвращаемые значения:

- Если при вызове DL=00h–0FEh, то ничего не возвращается.

Если при вызове DL=0FFh и была нажата клавиша, то ее код записывается в AL и сбрасывается флаг нуля (ZF=0).

Если при вызове DL=FFh и не была нажата клавиша, то ZF=0 и AL=00h.

Примечание. При использовании данной функции в случае ввода <Ctrl>+<C> или <Ctrl>+<Break> какого-либо специального действия не производится.

Для чтения расширенных ASCII-кодов необходимо обращаться к данной функции дважды. При первом обращении в AL возвращается значение 0, что указывает на наличие расширенного кода.

07h (INT 21h) — нефильтрованный ввод без эхо

Входные данные:

- AH=07h

Возвращаемые значения:

- AL = ASCII-код символа, полученный от стандартного устройства ввода.

Примечание. При использовании данной функции в случае ввода <Ctrl>+<C> или <Ctrl>+<Break> какого-либо специального действия не производится.

Для чтения расширенных ASCII-кодов необходимо обращаться к данной функции дважды. При первом обращении в AL возвращается значение 0, что указывает на наличие расширенного кода.

08h (INT 21h) — ввод символа без эхо

Входные данные:

- AH=08h

Возвращаемые значения:

- AL = ASCII-код символа или 0. Если AL=0, повторный вызов этой функции возвратит в AL расширенный ASCII-код символа.

Примечание. Если считывается комбинация клавиш <Ctrl>+<C> или <Ctrl>+<Break>, то выполняется прерывание INT 23h.

Для чтения расширенных ASCII-кодов необходимо обращаться к данной функции дважды. При первом обращении в AL возвращается значение 0, что указывает на наличие расширенного кода.

0Ah (INT 21h) — буферизированный ввод с клавиатуры

Входные данные:

- AH=0Ah

DS:DX=адрес входного буфера

Возвращаемые значения:

- В буфер помещается введенная строка.

Примечание. Перед вызовом этой функции необходимо подготовить буфер, в первом байте которого должно быть указано максимальное число вводимых символов от 1 до 254 (включая возврат каретки, CR), а остальное содержимое буфера можно использовать как подсказку для ввода.

После выполнения функции начиная с третьего байта буфер будет содержать символы введенной строки, включая последний символ возврата каретки 0Dh. В первом байте буфера будет максимальное число символов, определенное перед вызовом функции, а во втором байте буфера — фактическое число введенных символов (без кода возврата каретки).

Символы считываются до нажатия клавиши <Enter> или до тех пор, пока в буфер не будет помещено на один символ меньше, чем это определено максимальным числом символов. В последнем случае будет звучать сигнал на каждую очередную попытку ввода, пока не будет нажата клавиша <Enter>.

При вводе строки можно использовать клавиши редактирования:

<Esc> — начинает ввод сначала;

<F3> — выводит на экран подсказку для ввода;

<F5> — сохраняет текущую строку как подсказку;

<Backspace> — стирает предыдущий символ.

Если считывается комбинация клавиш <Ctrl>+<C> или <Ctrl>+<Break>, то выполняется прерывание INT 23h.

Кроме перечисленных, DOS предоставляет служебные функции для работы с клавиатурой:

0Bh (INT 21h) — проверить состояние ввода

Проверяет, имеется ли символ с устройства стандартного ввода.
Входные данные:

- AH=0Bh

DS:DX=адрес входного буфера

Возвращаемые значения:

- AL=00h — если символ отсутствует;
- AL=FFh — если имеется по крайней мере один символ.

Примечание. Эту функцию удобно использовать перед вызовом функций 01h, 07h и 08h, чтобы не ждать нажатия клавиши.

Если считывается комбинация клавиш <Ctrl>+<C> или <Ctrl>+<Break>, то выполняется прерывание INT 23h.

Пример использования:

```
....
mov ah,0bh      ; номер функции
int 21h         ; переход в MS-DOS
or al,al        ; есть ли символ?
jnz ready       ; перейти, если символ присутствует
...
```

0Ch (INT 21h) — очистить буфер и считать символ

Данная функция очищает буфер клавиатуры, затем вызывает функцию ввода, заданную в регистре AL, поэтому вызванная функция ввода будет ждать ввода с клавиатуры, а не использовать нажатый ранее и еще не обработанный символ. Часто функцию 0Ch используют для запроса согласия у пользователя перед выполнением действия, например: "Отформатировать диск Y/N?".

Входные данные:

- AH=0Ch
- AL=номер функции ввода, которая будет активирована после очистки буфера, допускаются следующие значения:

  01h — ввод с клавиатуры;

  06h — ввод с консоли;

  07h — нефильтрующий ввод без эхо-вывода;

  08h — ввод без эхо-вывода;

  0Ah — буферизированный ввод.

Возвращаемые значения:

- Зависит от вызванной функции

Примечание.

Наличие или отсутствие во время выполнения данной функции проверки кода <Ctrl>+<C> и <Ctrl>+<Break> зависит от номера функции, помещенного в регистр AL.

3Fh (INT 21h) — чтение из файла или устройства

Ввод с клавиатуры с помощью функции 3Fh прерывания INT 21h осуществляется точно так же, как и чтение из файла.

Обычно за стандартным устройством ввода (по умолчанию за клавиатурой) закреплен предопределенный дескриптор 0, который нужно поместить в регистр BX.

В CX нужно указать число вводимых символов (достаточно указать максимальную длину строки, например, 80 байт). Ввод завершается после нажатия клавиши <Enter>.

В регистр AX возвращается число реально введенных байтов, при этом учитываются также два байта (0Ah и 0Dh), поступающие во входной буфер от нажатия клавиши <Enter>.

Особая ситуация возникает, если попытаться ввести больше символов, чем затребовано функцией 3Fh. В процессе выполнения этой функции все вводимые символы тут же извлекаются из кольцевого буфера ввода и пересылаются в буфер DOS. Обнаружив во входном потоке коды клавиши <Enter>, DOS пересылает из этого буфера в буфер пользователя в программе точно затребованное число символов (естественно, без кодов <Enter>, которые располагаются в конце вводимой строки). Остальные символы остаются в буфере DOS, готовые к вводу. Фактически, если не принять специальных мер к очистке буфера, они поступят в программу при очередном запросе 3Fh, даже если оператор еще не начал вводить очередную порцию данных. Очевидно, что в этом случае будет нарушена синхронизация хода выполнения программы с работой оператора.

Пример:

```
buff db 80 dup(?)
fhandle dw 0

        mov ah,3fh
        mov bx,fhandle
        mov dx,offset buff
        mov cx,8
        int 21h
```

6.3.2. Функции BIOS

00h, 10h, 20h (INT 16h) — прочитать символ с клавиатуры с ожиданием

Входные данные:

- AH=00h (83/84-key), 10h (101/102-key), 20h (122-key)

Возвращаемые значения:

- AL=ASCII-код символа, 0 или префикс скан-кода
- AL=скан-код нажатой клавиши или расширенный ASCII-код

Примечание.

Если нажатой клавише соответствует ASCII-символ, то в AH возвращается код этого символа, а в AL — скан-код клавиши. Если нажатой клавише соответствует расширенный ASCII-код, в AL возвращается префикс скан-кода (например, 0E0h для клавиш дополнительной клавиатуры) или 0, если префикса нет, а в AH — расширенный ASCII-код.

01h, 11h, 21h (INT 16h) — проверка символа

Входные данные:

- AH=01h (83/84-key), 11h (101/102-key), 21h (122-key)

Возвращаемые значения:

- ZF=1, если буфер пуст
- ZF=0, если в буфере присутствует символ, тогда

AL= ASCII-код символа, 0 или префикс скан-кода

AH=скан-код нажатой клавиши или расширенный ASCII-код

02h, 12h, 22h (INT 16h) — считать состояние клавиатуры

Входные данные:

- AH=02h (83/84-key), 12h (101/102-key), 22h (122-key)

Возвращаемые значения:

- AL=байт состояния клавиатуры 1

  AH=байт состояния клавиатуры 2 (только для функций 12h и 22h)

Флаги состояния клавиатуры 1:
бит 0: правая клавиша Shift нажата
бит 1: левая клавиша Shift нажата
бит 2: клавиша Ctrl нажата
бит 3: клавиша Alt нажата
бит 4: переключатель Scroll Lock включен
бит 5: переключатель Num Lock включен
бит 6: переключатель Caps Lock включен
бит 7: переключатель Insert включен
Байт флагов клавиатуры 1 хранится в области ПЗУ BIOS по адресу 0000:0417h.

Флаги состояния клавиатуры 2:
бит 0: правая клавиша Shift нажата
бит 1: левая клавиша Shift нажата
бит 2: любая из клавиш Ctrl нажата
бит 3: любая из клавиш Alt нажата
бит 4: переключатель Scroll Lock включен
бит 5: переключатель Num Lock включен
бит 6: переключатель Caps Lock включен
бит 7: переключатель Insert включен
Байт флагов клавиатуры 1 хранится в области ПЗУ BIOS по адресу 0000:0417h.

В листинге 6.4 показан пример, в котором программа сначала запрашивает нажатие любой клавиши, затем просит нажать клавишу <Y>, затем клавишу <N> или <n>, затем сочетание клавиш <Shift+F2>. Программа не будет продолжать свою работу, пока пользователь не нажмет требуемую клавишу/клавиши. При этом в программе нажатие клавиши <Y> проверяется по ее ASCII-коду, а нажатие <N> или <n> по скан-коду клавиши.

Листинг 6.4. Пример обработки нажатия клавиш (inchar1.asm)

```
        .model tiny

        cr    equ   0Dh
        lf    equ   0Ah

        .code
        org     100h
start:
        mov     dx,offset str1
```

```
        call    subrout

        mov     ah,10h
        int     16h

        mov     dx,offset str2
        call    subrout
Key_Y:
        mov     ah,10h
        int     16h
        cmp     al,'Y'
        jne     Key_Y

        mov     dx,offset str3
        call    subrout
Key_Nn:
        mov     ah,10h
        int     16h
        cmp     ah,31h
        jne     Key_Nn

        mov     dx,offset str4
        call    subrout

Key_ShiftF2:
        mov     ah,10h
        int     16h
        cmp     al,0

        jnz     Key_ShiftF2
        cmp     ah,55h
        jnz     Key_ShiftF2

         mov    dx,offset str5
        call    subrout

        ret

str1    db    "Для продолжения нажмите любую клавишу...",cr,lf,'$'
str2    db    "Для продолжения нажмите клавишу 'Y'",cr,lf,'$'
str3    db    "Для продолжения нажмите клавишу 'N' или 'n'",cr,lf,'$'
str4    db    "Для продолжения нажмите сочетание Shift-F2",cr,lf,'$'
str5    db    "The End!",cr,lf,'$'

subrout PROC
        mov     ah,9
        int     21h
        ret
subrout ENDP

        end     start
```

В листинге 6.4 используются только функции посимвольного ввода, поэтому в листинге 6.5 показана программа, которая использует функцию DOS 0Ah (INT 21h), позволяющая за одну операцию ввести целую строку. Данная программа преобразует латинские буквы строки из нижнего регистра в верхний, и, наоборот — из верхнего регистра в нижний. Для этого просто над каждым ASCII-кодом символа строки выполняется операция XOR на значение 20h. Например, если над ASCII-кодом прописной буквы "F" (46h) выполнить операцию XOR 20h, то будет получен код 66h — это ASCII-код строчной буквы "f". Справедливо и обратное — если над ASCII-кодом строчной буквы "f" (66h) выполнить операцию XOR 20h, то будет получен ASCII-код прописной буквы "F" (46h). Такой простой операцией можно выполнять преобразование латинских букв из строчных в прописные и обратно.

Пример работы программы:

```
Введите строку для преобразования: Ivan Sklyaroff
Результат: iVAN sKLYAROFF
```

Листинг 6.5. Пример ввода и преобразования строки (inchar2.asm)

```
        .model tiny
        .code
        org     100h
start:
        mov     dx,offset mess1
        mov     ah,9
        int     21h

        mov     dx,offset buffer            ; считываем строку в буфер
        mov     ah,0Ah
        int     21h

        mov     dx,offset crlf
        mov     ah,9
        int     21h

        mov     dx,offset mess2
        mov     ah,9
        int     21h

        xor     di,di
        xor     cx,cx
        mov     cl,blength
        mov     si,cx                       ; в SI - длина буфера
        xor     ax,ax
next:
        mov     bl,byte ptr bcontents[di]   ; взять байт из буфера
        xor     bl,20h                      ; операция XOR 20h

        mov     dl,bl                       ; выводим символ на экран
        mov     ah,2
        int     21h

        inc     di                          ; следующий байт в буфере

        cmp     di,si           ; если счетчик меньше числа символов, то
        jb      next            ; продолжить
```

```
        ret

mess1   db      "Введите строку для преобразования:$"
mess2   db      "Результат:$"
crlf    db      0Dh, 0Ah,'$'   ; перевод строки
buffer  db      100            ; максимальный размер буфера
blengthdb       ?              ; здесь будет размер буфера после ввода
bcontents db    100 dup (?)    ; содержимое буфера

        end start
```

6.4. Работа с файлами

Система MS-DOS многое унаследовала от таких систем, как CP/M и UNIX. Возможно, именно поэтому в ней существует две группы функций для работы с файлами. Первая группа функций, совместима с древней системой CP/M и использует структуру FCB, а передачу данных осуществляет через структуру данных DTA. Структура FCB постоянно хранится в пространстве памяти, выделенной прикладной программе. Функции, использующие FCB, позволяют выполнять все основные операции с файлами (создание, открытие, закрытие, уничтожение, чтение, запись), но эти функции не поддерживают иерархическую (древовидную) файловую структуру, которая впервые была введена в MS-DOS версии 2, поэтому ими можно пользоваться для доступа к файлам, находящимся только в текущем подкаталоге текущего диска.

Вторая группа функций, совместимая с UNIX, использует в своей работе так называемые дескрипторы. Эти функции позволяют выполнять все основные операции с файлами, но они также позволяют передавать системе MS-DOS строку символов, завершающуюся нулем, в которой описывается расположение файла в иерархической файловой структуре (имя диска и путь), имя файла и расширение. При успешной операции открытия или создания файла система MS-DOS возвращает 16-битный дескриптор (16-битное число), который сохраняется программой и используется для последующих манипуляций с файлом. При этом системой инициализируются первые пять стандартных дескрипторов следующим образом:

0 (STDIN) — стандартное устройство ввода (как правило, клавиатура);

1 (STDOUT) — стандартное устройство вывода (как правило, экран монитора);

2 (STDERR) — устройство вывода сообщений об ошибках (всегда экран монитора);

3 (AUX) — последовательный порт (обычно COM1);

4 (PRN) — параллельный порт (обычно LPT1);

Функции, использующие дескриптор, полностью поддерживают иерархическую файловую структуру, поэтому позволяют программисту создавать, открывать, закрывать и уничтожать файлы в любом подкаталоге любого диска.

Функции FCB, кроме того, что не позволяют работать с файлами вне текущего каталога, очень не удобны и громоздки в использовании, поэтому они устарели сразу с выходом второй версии MS-DOS и практически с того момента программистами не использовались. Поэтому мы будем рассматривать только функции, работающие с файлами на основе дескриптора.

Ключевым в работе с файлами является понятие "открытие файла". Начать любую работу с файлом (чтение файла, запись в файл и пр.) можно только после того, как он открыт. При открытии файла ему системой MS-DOS назначается дескриптор, который представляет собой уникальное 16-битное число и служит своего рода меткой в дальнейшем при обращении к файлу. По завершении работы с файлом его нужно закрыть, при этом освобождается выделенный файлу дескриптор. Для открытия и закрытия файла имеются соответствующие функции DOS, о которых будет рассказано далее. Например, при чтении файла последовательность действий может выглядеть так: открытие файла, установка позиции чтения файла (если файл читается не сначала), чтение файла, закрытие файла.

Одновременно может быть открыто множество файлов. Устройства со стандартными дескрипторами, которые были перечислены выше, считаются открытыми постоянно.

Файл можно создать с помощью функции DOS 3Ch. Эта функция создает новый файл и сразу же открывает его. Если файл уже существует, функция 3Ch все равно открывает его, присваивая ему нулевую длину, иначе говоря, удаляет содержимое файла, если оно имелось. Если не нужно чтобы содержимое существующего файла уничтожалось, следует использовать вместо функции 3Ch функцию 5Bh.

На самом деле при записи в файл данные первоначально временно записываются во внутренний буфер системы MS-DOS, а во время закрытия файла сбрасываются на диск. Для принудительной записи из внутреннего буфера MS-DOS на диск можно использовать функцию 68h прерывания INT 21h (аналог функции fflush() в языке программирования Си) или функцию 0Dh прерывания INT 21h для критических участков программы.

Ниже перечислены практически все функции MS-DOS для работы с файлами и директориями, а в листинге 6.6 приведена программа, демонстрирующая работу с файлами.

6.4.1. Создание и открытие файлов

3Ch (INT 21h) — создать файл

Входные данные:

- AH=3Ch
- CX = атрибут файла (можно комбинировать биты):

  бит 0: файл только для чтения

  бит 1: скрытый файл

  бит 2: системный файл

  бит 3: метка тома (игнорируется функцией 3Ch)

  бит 4: директория (должен быть 0 для функции 3Ch)

  бит 5: атрибут архива

  бит 6: не используется

  бит 7: файл можно открывать разным процессам в Novell Netware

  биты 8-15: зарезервированы (0)
- DS:DX = адрес ASCIZ-строки с полным именем файла (строка ASCII-символов, оканчивающаяся нулем)

Возвращаемые значения:

- CF=0 и AX=дескриптор файла, если не произошла ошибка
- CF=1 и AX=03h, если путь не найден
- CF=1 и AX=04h, если слишком много открытых файлов
- CF=1 и AX=05h, если доступ запрещен

Примечание. Если файл уже существует, то он открывается и усекается до нулевой длины.

3Dh (INT 21h) — открыть существующий файл

Входные данные:

- AH=3Dh
- AL = режим доступа:

  бит 0: открыть для чтения

  бит 1: открыть для записи

  бит 2: открыть для чтение/запись

  биты 3: зарезервирован (0)

  бит 4-6: режим совместного использования:

000 – режим совместимости

001 – запрет чтения и записи другими программами

010 – блокировка записи другими программами

011 – запрет чтения другими программами

100 – полный доступ

111 – сетевой FCB (доступен в течение серверного вызова)

бит 7: флаг наследования: 0 – дочерний процесс наследует дескриптор; 1- дочерний процесс не наследует дескриптор.

- DS:DX = адрес ASCIIZ-строки с полным именем файла (строка ASCII-символов, оканчивающаяся нулем)

Возвращаемые значения:

- CF=0 и AX=дескриптор файла, если не произошла ошибка
- CF=1 и AX=03h, если путь не найден
- CF=1 и AX=04h, если слишком много открытых файлов
- CF=1 и AX=05h, если доступ запрещен
- CF=1 и AX=0Ch, недействительный режим доступа

5Bh (INT 21h) — создать и открыть существующий файл

Входные данные:

- AH=5Bh
- CX = атрибуты файла (можно комбинировать биты): такие же как в функции 3Ch
- DS:DX = адрес ASCIIZ-строки с полным именем файла (строка ASCII-символов, оканчивающаяся нулем)

Возвращаемые значения:

- CF=0 и AX=дескриптор файла, если не произошла ошибка
- CF=1 и AX=03h, если путь не найден
- CF=1 и AX=04h, если слишком много открытых файлов
- CF=1 и AX=05h, если доступ запрещен
- CF=1 и AX=50h, если файл уже существует

5Ah (INT 21h) — создать и открыть временный файл

Создает файл с уникальным именем в текущем или заданном каталоге.
Входные данные:

- AH=5Ah
- CX = атрибуты файла (можно комбинировать биты): такие же как в функции 3Ch
- DS:DX = адрес ASCIIZ-строки с полным именем файла (строка ASCII-символов, оканчивающаяся нулем)

Возвращаемые значения:

- CF=0 и AX=дескриптор файла, если не произошла ошибка
- CF=1 и AX=03h, если путь не найден
- CF=1 и AX=04h, если слишком много открытых файлов
- CF=1 и AX=05h, если доступ запрещен

6Ch (INT 21h) — создать или открыть файл с длинным именем

Входные данные:

- AH=6Ch
- AL=00h
- BX = режим доступа

биты 0-2: тип доступа:

000 – только чтение

001 – только запись

010 – чтение-запись

100 –только для чтения, не изменять время последнего обращения к файлу

биты 4-6: режим совместного использования (см. функцию 3Dh)

бит 7: наследование: 0 – дочерний процесс наследует дескриптор; 1 - дочерний процесс не наследует дескриптор.

биты 8: данные не буферизуются

бит 9: не архивировать файл, если используется архивирование файловой системы (Doublespace)

бит 10: использовать число в DI для записи в конце короткого имени файла

биты 11-12: зарезервированы (0)

бит 13: не вызывать прерывание 24h при критических ошибках

бит 14: сбрасывать буфера на диск после каждой записи в файл

- CX = атрибуты файла (можно комбинировать биты): такие же как в функции 3Ch
- DX = действие

биты 0-3: действие, если файл существует:

0000 – не выполняется,

0001 – открытие файла,

0010 – замена файла.

биты 4-7: действие, если файл не существует:

0000 – не выполняется

0001 – создание файла

биты 8-15: зарезервированы

- DX:SI = адрес ASCIIZ-строки с полным именем файла (строка ASCII-символов, оканчивающаяся нулем)
- DI = число, которое будет записано в конце короткого варианта имени файла

Возвращаемые значения:

CF=0

- AX=дескриптор файла

CX=1, если файл открыт

CX=2, если файл создан

CX=3, если файл заменен

CF=1, если произошла ошибка

AX=код ошибки (7100h, если функция не поддерживается)

6.4.2. Чтение и запись в файл

3Fh (INT 21h) — чтение из файла или устройства

Входные данные:

- AH=3Fh
- BX=дескриптор
- CX = число байтов
- DS:DX = адрес буфера для приема данных

Возвращаемые значения:

- CF=0 и AX=число считанных байтов, если не было ошибки
- CF=1 и AX=05h, если доступ запрещен

- CF=1 и AX=06h, если неправильный идентификатор

42h (INT 21h) — установить указатель чтения/записи

Входные данные:

- AH=42h
- BX=дескриптор
- CX:DX = смещение, на которое нужно переместить указатель (со знаком)
- AL=перемещение:

  0 – от начала файла

  1 – от текущей позиции

  2 – от конца файла

Возвращаемые значения:

- CF=0 и CX:DX=новое смещение указателя (в байтах от начала файла), если не произошла ошибка
- CF=1 и AX=06h, если неправильный идентификатор

Примечание. Данная функция используется также для определения длины файла – достаточно вызвать ее с CX=0, DX=0, AL=2, и в CX:DX будет возвращена длина файла в байтах.

40h (INT 21h) — записать в файл или на устройство

Входные данные:

- AH=40h
- BX=дескриптор
- CX=число байтов
- DS:DX = адрес буфера с данными

Возвращаемые значения:

- CF=0 и AX=число записанных байтов, если не произошла ошибка
- CF=1 и AX=05h, если доступ запрещен
- CF=1 и AX=06h, если неправильный дескриптор

*Примечание.*Если при вызове функции CX=0, то файл усекается или расширяется до текущего положения указателя файла.

68h (INT 21h) — сброс файловых буферов MS-DOS на диск

Принудительно переносит все данные из внутренних буферов системы MS-DOS, связанных с заданным дескриптором, на устройство.

Входные данные:

- AH=68h
- BX=дескриптор

Возвращаемые значения:

- CF=0, если функция успешно выполнена
- CF=1, если произошла ошибка (AX=код ошибки)

0Dh (INT 21h) — сброс всех файловых буферов на диск

Сбрасывает все файловые буферы. Все данные, записанные пользовательскими программами и временно буферизированные в MS-DOS, записываются физически на диск.

Входные данные:

- AH=0Dh

Возвращаемые значения:

- ничего не возвращается

Пример:

```
...
mov ah,0Dh ; номер функции
int 21h
...
```

Функция сбрасывает все внутренние буферы, связанные с файлом на диск, закрывает файл и освобождает для последующего использования дескриптор. Если файл изменялся, то обновляются дата и время создания, а также размер файла.

6.4.3. Закрытие и удаление файла

3Eh (INT 21h) — закрыть файл

Входные данные:

- AH=3Eh
- BX=дескриптор

Возвращаемые значения:

- CF=0, если функция успешно выполнена
- CF=1 и AX=6, если неправильный дескриптор

41h (INT 21h) — удалить файл

Удаляет файл из указанного каталога.

Входные данные:

- AH=41h
- DS:DX = адрес ASCIIZ-строки с полным именем файла

Возвращаемые значения:

- CF=0, если функция не выполнена
- CF=1 и AX=02h, если файл не найден

  CF=1 и AX=03h, если путь не найден

  CF=1 и AX=05h, если доступ запрещен

Примечание. Функция не выполняется, если:

— отсутствует хотя бы один элемент пути;

— указанный файл существует, но имеет атрибут "только для чтения" (проверить и изменить атрибут можно с помощью функции 43h прерывания INT 21h).

Функция 41h не позволяет использовать маски (символы * и ? в имени файла) для удаления сразу нескольких файлов, но начиная с DOS 7.0, обновленная функция способна работать с масками.

LFN 41h (INT 21h) — удалить файл с длинным именем[5]

Входные данные:

- AX=7141h
- DS:DX = адрес ASCIIZ-строки с длинным именем файла
- SI=0000h: маски не разрешены и атрибуты в CX игнорируются
- SI=0001h: маски в имени файла и атрибуты в CX разрешены:

[5] LFN-функции — это функции MS-DOS, работающие с длинными именами файлов (LFN, Long File Names — Длинные Имена Файлов), длиной до 255 символов. Этот набор функций впервые появился в DOS 7.0 вместе с Windows 95. Ранее функции MS-DOS работали только с файлами формата 8.3 (восемь символов имени и три символа расширения).

CL=атрибуты, которые файлы могут иметь

CH=атрибуты, которые файлы должны иметь

Возвращаемые значения:

- CF=0, если файл или файлы удалены
- CF=1 и AX=код ошибки, если произошла ошибка. Код 7100h означает, что функция не поддерживается.

6.4.4. Поиск файлов

Поиск файлов с короткими именами на диске осуществляется с помощью двух функций: 4Eh (найти первый файл) и 4Fh (найти следующий файл).

Поиск файлов с длинными именами на диске осуществляется тремя функциями: найти первый файл, найти следующий файл, прекратить поиск.

4Eh (INT 21h) — найти первый файл

Входные данные:

- AH=4Eh
- CX = атрибуты поиска (биты могут комбинироваться):

  бит 0: только для чтения

  бит 1: скрытый файл

  бит 2: системный файл

  бит 3: метка тома

  бит 4: директория

  бит 5: атрибут архива

  биты 6-15: все остальные биты не используются
- DS:DX=адрес ASCIIZ-строки с именем файла, которое может включать путь и маски для поиска (символы * и ?)

Возвращаемые значения:

- CF=0 и область DTA заполняется данными, если файл найден
- CF=1 и AX=02h, если файл не найден; 03h, если путь не найден; 12h, если неправильный режим доступа.

4Fh (INT 21h) — найти следующий файл

Входные данные:

- AX=4Fh
- DTA – содержит данные от предыдущего вызова функции 4Eh и 4Fh.

Возвращаемые значения:

- CF=0 и DTA содержит данные о следующем найденном файле, если не произошла ошибка.
- CF=1 и AX=код ошибки, если произошла ошибка.

LFN 4Eh (INT 21h) — найти первый файл с длинным именем

Входные данные:

- AX=714Eh
- CL=атрибуты, которые файл может иметь (биты 0 и 5 игнорируются)
- CH= атрибуты, которые файл должен иметь
- SI=0: использовать Windows-формат даты/времени
- SI=1: использовать DOS-формат даты/времени

- DS:DX=адрес ASCIIZ-строки с маской поиска (может включать * и ?. Для совместимости маска *.* ищет все файлы в директории, а не только файлы содержащие точку в имени)
- ES:DI=адрес 318-байтного буфера для информации о файле.

Возвращаемые значения:

- CF=0
- AX=поисковый идентификатор
- CX=Unicode-флаг:

  бит 0: длинное имя содержит подчеркивания вместо преобразуемых Unicode-символов

  бит 1: короткое имя содержит подчеркивания вместо непреобразуемых Unicode-символов
- CF=1, AX=код ошибки, если произошла ошибка (7100h – функция не поддерживается)

Если файл найден, область данных по адресу ES:DI заполняется следующим образом:

00h: 4 байта – атрибуты файла

биты 0-6: атрибуты файла DOS

бит 8: временный файл

04h: 8 байт – время создания файла

0Ch: 8 байт – время последнего доступа к файлу

14h: 8 байт – время последней модификации файла

1Ch: 4 байта – старшее двойное слово длины файла

20h: 4 байта – младшее двойное слово длины файла

24h: 8 байта – зарезервировано

2Ch: 260 байт – ASCIIZ-имя файла длинное

130h: 14 байт – ASCIIZ-имя файла короткое

Дата создания/доступа/модификации записываются в одном из двух форматов, в соответствии со значением SI при вызове функции. Windows-формат – 64-битное число 100-наносекундных интервалов с 1 января 1601 года; если используется DOS-формат – в старшее двойное слово записывается DOS-дата, а в младшее – DOS-время.

LFN 4Fh (INT 21h) — найти следующий файл

Входные данные:

- AX=714Fh
- BX=поисковый идентификатор (от функции 4Eh)
- SI=формат даты/времени
- ES:DI=адрес буфера для информации о файле

Возвращаемые значения:

- CF=0 и CX=Unicode-флаг; если следующий файл найден
- CF=1 и AX=код ошибки, если произошла ошибка (7100h – функция не поддерживается)

LFN A1h (INT 21h) — закончить поиск файла

Входные данные:

- AX=7141h
- BX=поисковый идентификатор

Возвращаемые значения:

- CF=0, если операция выполнена

- CF=1 и AX=код ошибки, если произошла ошибка (7100h – функция не поддерживается)

6.4.5. Управление директориями

39h (INT 21h) — создать директорию

Создает директорию по указанному пути.
Входные данные:

- AX=39h
- DS:DX=адрес ASCIIZ-строки с путем, в котором все директории, кроме последней, существуют.

Возвращаемые значения:

- CF=0, если директория создана
- CF=1 и AX=3, если путь не найден; 5, если доступ запрещен

LFN 39h (INT 21h) — создать директорию с длинным именем

Входные данные:

- AX=7139h
- DS:DX=адрес ASCIIZ-строки с путем

Возвращаемые значения:

- CF=0, если директория создана
- CF=1 и AX=код ошибки (7100h, если функция не поддерживается)

3Ah (INT 21h) — удалить директорию

Входные данные:

- AX=3Ah
- DS:DX=адрес ASCIIZ-строки с путем, где последняя директория будет удалена (только если она пустая)

Возвращаемые значения:

- CF=0, если директория удалена
- CF=1 и AX=3, если путь не найден; 5, если доступ запрещен; 10h, если удаляется директория - текущая

LFN 3Ah (INT 21h) — удалить директорию с длинным именем

Входные данные:

- AX=713Ah
- DS:DX=адрес ASCIIZ-строки с путем

Возвращаемые значения:

- CF=0, если директория удалена
- CF=1 и AX=код ошибки

47h (INT 21h) — определить текущую директорию

Входные данные:

- AH=47h
- DL=номер диска (00h – текущий, 01h – A и т. д.)
- DS:SI = 64-байтный буфер для текущего пути (ASCIIZ-строка без имени диска, первого и последнего символа \)

Возвращаемые значения:

- CF=0 и AX=0100h, если операция выполнена
- CF=1 и AX=0Fh, если указан несуществующий диск

LFN 47h (INT 21h) — определить текущую директорию с длинным именем

Входные данные:

- AH=7147h
- DL=номер диска
- DS:SI = буфер для пути (ASCIIZ-строка без имени диска, первого и последнего символа \. Необязательно содержит лишь длинные имена – возвращается тот путь, который использовался при последней смене текущей директории.)

Возвращаемые значения:

- CF=0 , если директория определена
- CF=1 и AX=код ошибки

3Bh (INT 21h) — сменить директорию

Входные данные:

- AH=3Bh
- DS:DX = адрес 64-байтного ASCIIZ-буфера с путем, который станет текущей директорией

Возвращаемые значения:

- CF=0, если директория изменена
- CF=1 и AX=3, если путь не найден

LFN 3Bh (INT 21h) — сменить директорию с длинным именем

Входные данные:

- AH=713Bh
- DS:DX = адрес ASCIIZ-буфера с путем

Возвращаемые значения:

- CF=0, если директория изменена
- CF=1 и AX=код ошибки

Перед работой с любыми функциями LFN следует один раз вызывать подфункцию 0A0h, чтобы определить размеры буферов для имен файлов и путей.

Кроме того, при вызове любой функции LFN следует установить CF в 1 для совместимости с ранними версиями DOS. Старые версии DOS не изменяли CF, так что в результате, если функция не поддерживается, CF останется равным 1.

В листинге 6.6 приведена программа, демонстрирующая работу с файлами. Она создает на диске файл, делает в него запись и закрывает файл. Затем открывает созданный файл, перемещает позицию чтения/записи на пятый байт файла, считывает участок строки с пятого байта до конца строки, выводит его на экран, после чего закрывает файл. Файл, с которым работает программа текстовый. Он выбран только для наглядности. Принципы работы с другими типами файлов ничем не отличаются от принципов работы с текстовыми.

После выполнения любой функции для работы с файлами и директориями в случае ошибки устанавливается флаг CF=1 (см. описание каждой функции выше). Поэтому в программе проверяется этот флаг после вызова каждой функции с помощью инструкции JC, если флаг установлен, выводится ошибка. Это самый простой подход, лучше, чтобы программа анализировала код ошибки и выдавала более подробную информацию об ошибке.

Листинг 6.6. Программа, демонстрирующая работу с файлами (dosfile.asm)

```
        .model tiny
        .code
        org     100h
start:
        ; Создание файла
        mov     ah,03Ch
        mov     cx,0
        mov     dx,offset FileName
        int     21h

        jc      error

        ; Сохранение файлового дескриптора
        mov     [FileNumber],ax

        ; Запись строки в файл
        mov     ah,040h
        mov     bx,[FileNumber]
        mov     cx,12
        mov     dx,offset TextLine
        int     21h

        jc      error

        ; Закрытие файла
        mov     bx,[FileNumber]
        mov     ah,03Eh
        int     21h

        ; Открытие файла для чтения
        mov     ax,03D00h
        mov     dx,offset FileName
        int     21h

        jc      error

        ; Сохранение файлового дескриптора
        mov     [FileNumber],ax

        ; Установка позиции на пятый байт файла
        mov     ax,04200h
        mov     bx,[FileNumber]
        mov     cx,0
        mov     dx,5
        int     21h

        jc      error

        ; Чтение файла в специально отведенное место
        mov     ah,03Fh
```

```
        mov     bx,[FileNumber]
        mov     cx,5
        mov     dx,offset TextPlace
        int     21h

        jc      error

        ; Вывод прочитаных байт на экран
        mov     dx,offset TextPlace
        mov     ah,9
        int     21h

        ; Закрытие файла
        mov     bx,[FileNumber]
        mov     ah,03Eh
        int     21h

        ret
error:
        mov     dx,offset TextError
        mov     ah,9
        int     21h
        ret

FileName        db      "skl.txt",0
FileNumber      dw      ?
TextLine        db      "ABCDEFGHIJ",0Dh,0Ah
TextPlace       db      10 DUP (?),'$'
TextError       db      "Ошибка!",0Dh,0Ah,'$'

        end     start
```

6.5. Прерывания

Мы в предыдущих примерах часто использовали команду INT и говорили, что она вызывает обработчик прерывания. Настала пора узнать, что конкретно делает команда INT, и что такое прерывания и их обработчики.

Представьте себе такую ситуацию. Домработница убирает квартиру, и в это время раздается звонок телефона, одновременно стук в дверь и начинает "свистеть" закипевший чайник на кухне. Домработнице надо решить, что сделать в первую очередь: открыть дверь, выключить чайник или ответить на звонок. Она может, например, совсем не ответить на телефонный звонок, потому что вы ей приказали не отвечать на телефонные звонки.

Эта ситуация является прямой аналогией работы процессора. Во время выполнения программы процессор, так же как и домработницу, постоянно отвлекают различные события (от внешних устройств или программ), требующие его реакции. Временное прекращение обработки программы процессором для выполнения затребованных действий называют прерыванием. Прерывания делят на аппаратные и программные.

Аппаратные прерывания возникают в результате подачи сигнала от какого-либо устройства компьютера (клавиатура, системный таймер, жесткий диск и т. д.). Так как аппаратные прерывания возникают в случайные моменты времени, то их иногда называют еще асинхронными.

Программные прерывания вызываются искусственно с помощью команды INT или ее аналогов (int3, into). Эти прерывания являются синхронными.

6.5.1. Внутренние и внешние аппаратные прерывания

Аппаратные прерывания делятся на внутренние и внешние.

Внутренние аппаратные прерывания возникают внутри самого процессора, а для приема внешних у процессора есть два физических контакта NMI (NonMaskable Interrupt — немаскируемые прерывания) и INTR (INTerrupt Request — запрос прерывания) (рис. 6.2).

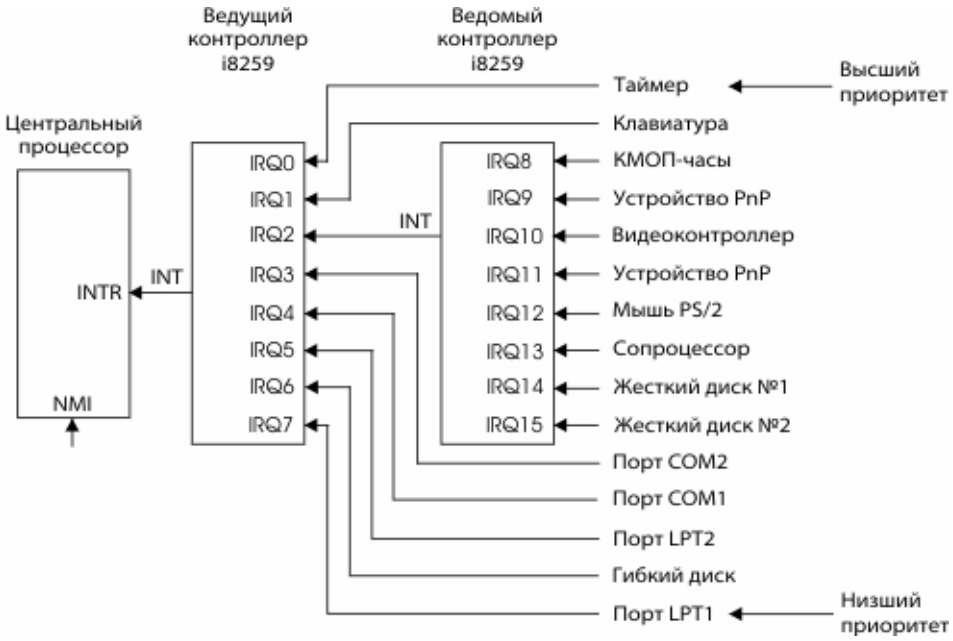

Рис. 6.2. Подсистема прерываний компьютера на базе процессора Intel

В зависимости от двух входов процессора внешние аппаратные прерывания делятся на маскируемые и немаскируемые.

Маскируемые прерывания — это те, которые можно запретить, т. е. программист в своей программе может указать процессору, что прерывания этого типа обслуживать не нужно. Маскируемые прерывания поступают на вход INTR процессора и связаны с нормальной работой устройств.

Немаскируемые прерывания запретить нельзя, они должны быть обслужены процессором сразу. Немаскируемые прерывания возникают в результате чрезвычайной ситуации в работе компьютера, например, сбоя в памяти или временном понижении напряжения. Немаскируемые прерывания поступают на вход NMI процессора.

Внутренние аппаратные прерывания называют еще исключениями. Они возбуждаются внутренними схемами самого процессора при возникновении одной из специально оговоренных ситуаций, например, при выполнении операции деления на ноль или при попытке выполнить несуществующую команду.

При возникновении прерывания (аппаратного или программного) процессор прекращает выполнение обрабатываемой программы и переключается на процедуру обработки прерывания. После того как данная процедура выполнит необходимые действия, прерванная программа продолжит выполнение с того места, где было приостановлено ее выполнение.

Перед тем как вызывать процедуру обработки прерывания процессор автоматически сохраняет в стеке регистры CS, IP и FLAGS\EFLAGS, чтобы прерванная программа после обработки прерывания смогла безболезненно продолжить свое выполнение. Регистры CS:IP содержат адрес команды, с которой необходимо начать выполнение после возврата из процедуры обработки прерывания, а FLAGS\EFLAGS — состояние флагов до момента передачи управления процедуре обработке прерывания. Если

необходимо сохранение других регистров до вызова обработчика прерываний, программист это должен обеспечивать сам в своей программе.

Помните, ранее отмечалось, что если exe-программа не использует стек, то его необходимо все равно задать. Теперь должно быть понятно, что стек используется при прерываниях, которые происходят при выполнении программы.

Разумеется, на разное прерывание нужно реагировать по-разному, т. е. для каждого типа прерывания должна быть своя процедура обработки прерывания.

В реальном режиме процессора допускается всего 256 источников прерываний, они нумеруются от 0 до 255. Для каждого прерывания существует уже готовая процедура обработки, и все эти процедуры включены в состав MS-DOS. Адреса этих процедур размещены в так называемой таблице векторов прерываний, соответственно вектором прерывания называют адрес процедуры обработки прерывания. Таблица векторов прерываний размещается в первом килобайте оперативной памяти и занимает адреса от 00000h до 00400h. Каждый адрес в таблице занимает двойное слово (4 байта): 1-е слово — это номер сегмента памяти, в котором находится соответствующая процедура обработки прерывания, а 2-е слово — смещение внутри этого сегмента. Прерыванию с номером 0 соответствует адрес 0000:0000, прерыванию с номером 1 – 0000:0004 и т. д.

Таким образом, смещение по которому находится вектор с номером N в таблице векторов прерываний, определяется как 4*N, а полный размер таблицы векторов прерываний составляет 4*256=1024 байт (1 Кбайт).

Как используется вектора и таблица векторов прерываний? Когда возникает прерывание, процессор по номеру источника прерывания путем умножения на 4 определяет смещение в таблице векторов прерываний. Затем сохраняет в стеке регистры CS, IP и FLAGS\EFLAGS, о чем мы говорили выше, и помещает первое слово вектора в регистр ip, а второе слово в регистр CS, и передает управление по адресу CS:IP, — так передается управление на процедуру обработки прерывания. Процедура обработки прерывания во время своего выполнения также может быть прервана, например, поступлением запроса от более приоритетного прерывания.

Когда процедура обработки прерывания закончит свою работу, она должна возобновить работу прерванной программы. Для этого ей надо восстановить из стека сохраненные там ранее значения IP, CS и FLAGS\EFLAGS. Делается это с помощью команды IRET, которая не имеет операндов. Команда IRET так и называется "возврат из обработчика прерываний", она самостоятельно загружает из стека значения IP, CS и FLAGS. Поэтому командой IRET все процедуры обработки прерываний завершают свою работу.

После возвращения из обработчика прерывания прерванная программа продолжит свою работу так, как будто и не было никакого прерывания.

Маскируемые прерывания делятся по уровням приоритетов. При одновременном поступлении нескольких запросов на прерывание, всегда обрабатывается прерывание с максимальным приоритетом. Если во время обработки прерывания приходит еще один запрос на прерывание с меньшим приоритетом, то он становится в очередь, а если с большим, то текущее прерывание само прерывается и ожидает, пока не будет обработано прерывание с более высоким уровнем приоритета.

Уровни приоритетов обозначаются сокращенно IRQ0-IRQ15. Аббревиатура IRQ происходит от английских слов Interrupt Request — запрос на прерывание. Самый высокий приоритет у IRQ 0, а самый низкий у IRQ 15. Физически, IRQ представляют собой отдельно проложенные линии (проводники) на материнской плате и соответствующие этим линиям контакты в интерфейсах устройств. Линии IRQ предназначены только для передачи запросов прерывания.

В табл. 6.2 приведены все аппаратные прерывания, расположенные в порядке убывания приоритета.

Таблица 6.2. *Аппаратные прерывания, расположенные в порядке убывания приоритета*

Прерывание	**Номер вектора**	**Адрес вектора**	**Источник сигнала прерывания**
IRQ0	INT 8h	0000:0020h	прерывание системного таймера, вызывается 18,2 раза в секунду

Таблица 6.2. *(окончание)*

IRQ1	INT 9h	0000:0024h	прерывание от клавиатуры, вызывается при каждом нажатии и отпускании клавиши
IRQ2	INT 0Ah	0000:0028h	к этому выходу подключается ведомый контроллер (прерывания IRQ8-IRQ15)
IRQ8	INT 70h	0000:01C0h	прерывание от часов реального времени
IRQ9	INT 0Ah или INT 71h	0000:01C4h	прерывание обратного хода луча, вызывается некоторыми адаптерами при обратном ходе луча
IRQ10	INT 72h	0000:01C8h	используется дополнительными устройствами
IRQ11	INT 73h	0000:01CCh	используется дополнительными устройствами
IRQ12	INT 74h	0000:01D0h	прерывание от мыши PS/2-типа
IRQ13	INT 02h или INT 75h	0000:01D4h	прерывание от математического сопроцессора. По умолчанию это прерывание отключено как на FPU, так и на контроллере прерываний
IRQ14	INT 76h	0000:01D8h	прерывание от первого IDE-контроллера "операция завершена"
IRQ15	INT 77h	0000:01DCh	прерывание от второго IDE-контроллера "операция завершена"
IRQ3	INT 0Bh	0000:002Ch	прерывание от последовательного порта COM2, вызывается, если порт COM2 получил данные
IRQ4	INT 0Ch	0000:0030h	прерывание от последовательного порта COM1, вызывается, если порт COM1 получил данные
IRQ5	INT 0Dh	0000:0034h	прерывание от параллельного порта LPT2
IRQ6	INT 0Eh	0000:0038h	прерывание от дисковода "операция завершена"
IRQ7	INT 0Fh	0000:003Ch	прерывание от параллельного порта LPT1

Вы можете посмотреть прерывания своего компьютера. Выберите "Пуск – Выполнить", наберите "msinfo32". В появившимся окне "Сведения о системе", слева, выберите вкладку "Ресурсы аппаратуры – Прерывания IRQ". В современных компьютерах количество IRQ может быть больше 15.

Программист может в своей программе запретить (замаскировать) обработку всех или любого сочетания маскируемых прерываний, кроме того, программист может заменить обработчик прерывания на свой. Далее рассказано, как выполнить все эти действия.

6.5.2. Запрет всех маскируемых прерываний

Для запрета всех маскируемых прерываний в процессоре имеется инструкция CLI. На самом деле эта инструкция просто сбрасывает флаг IF в 0.

В процессоре также имеется команд STI, которая устанавливает флаг IF в 1, тем самым, отменяя действие команды CLI.

6.5.3. Запрет определенного маскируемого прерывания

Как ведущий, так и ведомый контроллеры i8259 содержат специальный внутренний регистр маски прерываний IMR.

Для того чтобы запретить определенное прерывание, необходимо соответствующий бит в регистре IMR установить в 1. Номера разрядов регистра IMR соответствуют номерам линий сигналов прерывания, например IRQ0 и IRQ8 соответствует самый младший разряд в регистре IMR, а IRQ7 и IRQ15 — самый старший.

Регистр маски прерываний ведущего контроллера доступен для записи и считывания через порт 21h, а ведомого контроллера — через порт 0A1h.

Можно сразу записать байт с установленным битом в регистр IMR, однако, если в регистре IMR уже были установлены какие-нибудь биты, то мы можем затереть эту информацию. Поэтому принято сначала считывать содержимое регистра IMR, изменять нужный бит в считанном значении, а затем полученное значение помещать обратно в регистр маски прерываний.

Между двумя обращениями к одному и тому же порту рекомендуется вставлять циклы задержки, поскольку контроллер i8259 по сравнению с процессором работает слишком медленно, обычно для этого используется пара операторов:

```
jmp short $+2
jmp short $+2
```

Например, вот так можно запретить прерывание от клавиатуры (IRQ1):

```
in al,21h
or al,00000010b
jmp short $+2
jmp short $+2
out 21h,al
```

Программист должен предусмотреть в своей программе обратную операцию — демаскирование запрещенного прерывания. Например, демаскирование прерывания от клавиатуры будет выглядеть следующим образом:

```
in al,21h
and al,11111101b
jmp short $+2
jmp short $+2
out 21h,al
```

6.5.4. Собственный обработчик прерывания

Обработчик прерывания оформляется как обычная подпрограмма:

```
int_hook   proc   far
       ; тело обработчика
       iret
int_hook   endp
```

После того как обработчик написан, его можно привязать к номеру прерывания. Это можно сделать, записав адрес обработчика в таблицу векторов прерываний. Однако перед этим следует сохранить адрес предыдущего обработчика (который мы заменяем) чтобы по окончании работы программы можно было его восстановить, — это необходимо для того, чтобы не нарушить нормальную работу системы.

В листинге 6.7 показан пример корректной замены обработчика прерывания. Наш обработчик будет вызваться командой INT 77h.

Листинг 6.7. Корректная замена обработчика прерывания

```
push 0 ; Записываем сегментный адрес таблицы векторов прерываний
pop es ; в регистр ES
; Копируем адрес предыдущего обработчика в переменную old_hook
mov eax,dword ptr es:[77h*4]
mov dword ptr old_hook,eax
; Устанавливаем наш обработчик
pushf ; Помещаем регистр флагов в стек
cli ; Запрещаем прерывания, чтобы не произошло прерывания в момент
изменения адреса, т. к. возможен крах системы
```

```
; обработчик прерывания
; Помещаем дальний адрес нашего обработчика int_hook в таблицу
; векторов прерываний, в элемент с номером 77h
mov word ptr es:[77h*4], offset int_hook
mov word ptr es:[77h*4+2], seg int_hook
popf ; восстанавливаем исходное значение флага IF
; Тело программы
....
....
....
; Восстанавливаем предыдущий обработчик
push 0
pop es
pushf
cli
mov eax, word ptr old_hook
mov word ptr es:[77h*4],eax
popf
```

Существует еще один способ замены обработчика прерывания — с помощью функций операционной системы DOS 25h и 35h, которые вызываются по прерыванию int 21h:

Функция 35h (INT 21h) — получить вектор прерываний

Входные данные:

- AH=35h
- AL=номер прерывания (от 00h до 0FFh)

Возвращаемые значения:

- ES:BX=адрес обработчика прерывания для INT AL.

Функция 25h (INT 21h) — установить вектор прерываний

Входные данные:

- AH=25h
- AL= номер прерывания (от 00h до 0FFh)
- DS:DX=адрес обработчика прерываний

Возвращаемые значения:

- Функция ничего не возвращает.

В листинге 6.8 показан пример замены обработчика прерывания с помощью функций DOS.

Листинг 6.8. Замена обработчика прерывания с помощью функций DOS

```
; Копируем адрес предыдущего обработчика в переменную old_hook
mov ax,3577h
int 21h

mov word ptr old_hook,bx
mov word ptr old_hook+2,es

mov ax,2577h
mov dx,seg int_hook
```

```
mov ds,dx
mov dx,offset int_hook
int 21h
...
...
...

lds dx,old_hook
mov ah,25h
mov al,63h
int 21h
```

6.5.5. Распределение номеров прерываний

Как уже говорилось, в памяти отводится место под 256 векторов прерываний (1 Кбайт). Некоторые из них практически не используются; к другим, наоборот, обращается каждая программа.

В табл. 6.3 показана таблица векторов прерываний, чтобы вы имели представление о ее составе.

Таблица 6.3. *Таблица векторов прерываний*

Номер вектора	Адрес вектора	Назначение вектора
00h	00000h	Внутренне прерывание генерируется CPU при попытке деления на нуль
01h	00004h	Внутреннее прерывание пошагового выполнения программы. Генерируется после выполнения каждой машинной команды, если в слове флагов установлен бит пошаговой трассировки TF. Используется отладчиками
02h	00008h	Аппаратное немаскируемое прерывание. Обычно генерируется при ошибке четности в оперативной памяти и при запросе прерывания от сопроцессора
03h	0000Ch	Программное прерывание для трассировки. Генерируется при выполнении однобайтовой машинной команды с кодом CCh и обычно используется отладчиками для установки точки прерывания
04h	00010h	Внутреннее прерывание генерируется, когда арифметический результат приводит к переполнению
05h	00014h	Аппаратное прерывание от нажатия клавиши PrintScreen (печать экрана)
06h	00018h	Внутреннее прерывание недопустимого кода операции или длина команды больше 10 байт
07h	0001Ch	Особый случай отсутствия арифметического сопроцессора
08h	00020h	IRQ0 — аппаратное прерывание от системного таймера (генерируется 18,2 раза в секунду)
09h	00024h	IRQ1 — аппаратное прерывание от клавиатуры. Генерируется при каждом нажатии и отпускании клавиши. Используется для чтения данных из клавиатуры
0Ah	00028h	IRQ2 — используется для каскадирования аппаратных прерываний
0Bh	0002Ch	IRQ3 — аппаратное прерывание от последовательного порта COM2

Таблица 6.3. *(продолжение)*

0Ch	00030h	IRQ4 — аппаратное прерывание от последовательного порта COM1
0Dh	00034h	IRQ5 — аппаратное прерывание от параллельного порта LPT2
0Eh	00038h	IRQ6 — аппаратное прерывание от гибкого диска. Генерируется контроллером НГМД после завершения операции ввода/вывода
0Fh	0003Ch	IRQ7 — аппаратное прерывание от параллельного порта LPT1
10h	00040h	Программы BIOS обслуживания видеосистемы
11h	00044h	Определение конфигурации устройств в системе
12h	00048h	Определение размера оперативной памяти
13h	0004Ch	Программы BIOS обслуживания дисков
14h	00050h	Программы BIOS обслуживания последовательного порта
15h	00054h	Расширенный сервис
16h	00058h	Программы BIOS обслуживания клавиатуры
17h	0005Ch	Программы BIOS обслуживания принтера
18h	00060h	Запуск BASIC в ПЗУ, если он есть
19h	00064h	Перезагрузка операционной системы
1Ah	00068h	Программы BIOS обслуживания часов реального времени
1Bh	0006Ch	Обработчик прерывания, возникающего, если пользователь нажал комбинацию клавиш <Ctrl+Break>
1Ch	00070h	Вектор для прикладной обработки прерываний от системного таймера
1Dh	00074h	Адрес таблицы видеопараметров, используемой программами BIOS
1Eh	00078h	Адрес таблицы параметров дискеты, используемой программами BIOS
1Fh	0007Ch	Указатель на графическую таблицу для символов с кодами ASCII 128-255
21h	00084h	Диспетчер функций DOS
22h	00088h	Адрес возврата в родительский процесс после завершения текущего
23h	0008Ch	Адрес обработчика системы DOS, активизируемого нажатием клавиш <Ctrl>+<C>
24h	00090h	Адрес обработчика DOS критических ошибок
25h	00094h	Активизирует службу DOS для чтения данных с диска по абсолютным адресам
26h	00098h	Активизирует службу DOS для записи данных с диска по абсолютным адресам
27h	0009Ch	Завершает работу программы, но оставляет ее в памяти под наблюдением DOS
2Fh	000BCh	Программное прерывание, используемое для связи с резидентными программами
33h	000CCh	Программы обслуживания мыши, реализуемые в драйвере мыши
43h	0010Ch	Указывает набор графических знаков (EGA, PS/2)

Таблица 6.3. *(окончание)*

4Ah	00128h	Вектор для прикладной обработки прерываний от будильника (часов реального времени)
60h...67h	00180h...0019Ch	Свободные векторы для использования прикладными программами
68h...6Fh	001A0h...001BCh	Не используются
70h	001C0h	IRQ8 — аппаратное прерывание от будильника (часов реального времени)
71h	001C4h	IRQ9 — аппаратное прерывание от подключенной к компьютеру нестандартной аппаратуры
72h	001C8h	IRQ10 — используется дополнительными устройствами
73h	001CCh	IRQ11 — используется дополнительными устройствами
74h	001D0h	IRQ12 — прерывание от мыши PS/2-типа
75h	001D4h	IRQ13 — аппаратное прерывание от арифметического сопроцессора
76h	001D8h	IRQ14 — аппаратное прерывание от первого IDE-контролера "операция завершена"
77h	001DCh	IRQ15 — аппаратное прерывание от второго IDE-контролера "операция завершена"
78h...7F	001DCh...001FCh	Зарезервировано
80h...85h	00200h...00214h	Зарезервировано для BASIC
86h...F0h	00218h...003C0h	Используются интерпретатором BASIC
F1h...FFh	003C4h...003FCh	Зарезервировано для временного использования программистом

Большинство векторов прерываний содержат адреса обработчиков прерываний. Однако некоторые номера прерываний просто указывают на различную служебную информацию. Например, прерывание 1Eh содержит адрес, по которому хранятся параметры инициализации дисковода для дискет; вектор прерывания 1Fh указывает на битовые комбинации, используемые BIOS при выводе на экран дисплея знаков текста, а прерывания 41h и 46h указывают на параметры жесткого диска.

Такие векторы прерываний не используются для выполнения прерываний. Если, к примеру, вы попытаетесь выполнить прерывание 1Eh, то, скорее всего, вызовите аварийный сбой в работе программы, т. к. вектор 1Eh указывает на данные, а не на выполняемый код.

ДЕНЬ 7

Программирование под Windows

Программирование под Windows существенно отличается от программирования под MS-DOS, однако все основные команды и конструкции ассемблера, которые мы изучили под MS-DOS, будут точно также работать и под Windows.

7.1. Особенности программирования под Windows

В чем-то программирование под Windows даже проще чем под DOS.

Под Windows программисту не нужно беспокоиться о моделях памяти и сегментах, т. к. используется только одна плоская модель памяти FLAT. То есть в Windows нет больше сегментов по 64 Кбайт, а память представляется одним большим последовательным виртуальным пространством размером 4 Гбайт. Любые сегментные регистры программист может использовать для адресации к любой точке памяти в пределах 4 Гбайт.

Все приложения под Windows должны пользоваться только функциями API (Application Program Interface — программный интерфейс приложения) предоставляемыми этой системой. Взаимодействие с внешними устройствами и ресурсами операционной системы осуществляется посредством этих функций. И хотя существует возможность в программах под Windows задействовать функции BIOS, а также обращаться напрямую к устройствам через порты ввода-вывода, но это не относится к числу стандартных способов программированием под Windows.

Последние версии Windows содержат свыше 3000 API-функций. Функции API подключаются к программе с помощью динамических библиотек (DLL, Dynamic Link Library), которые хранятся в системном каталоге \Windows\System32\. Основными такими библиотеками являются:

- kernel32.dll — содержит API-функции управления памятью, процессами и потоками.
- user32.dll — отвечает за систему окон с подсистемой сообщений.
- gdi32.dll — библиотека интерфейса графического устройства (Graphic Device Library), содержит функции рисования.

Так как ОС Windows почти полностью написана на языке программирования Си, то и функции API и все сопутствующие им структуры имеют синтаксис свойственный этому языку. Поэтому для лучшего понимания программирования под Windows очень желательно знание языка Си. Но я постараюсь объяснить так, чтобы все было понятно и без знания Си.

Т. к. Windows имеет графический пользовательский интерфейс (GUI, Graphic User Interface), то основными прикладными программами под Windows являются оконные графические приложения (приложения GUI). Однако Windows позволяет создавать и консольные приложения для работы с командной строкой. Как графические, так и консольные приложения под Windows должны задействовать соответствующие API-функции.

В Windows имеются еще два вида приложений, которые не являются ни консольными, ни графическими и требуют особого подхода при программировании,

их можно назвать системными приложениями — это службы (services) и драйверы (drivers).

Каркас программы на ассемблере под Windows выглядит почти точно также как под DOS:

```
.386P
.model flat,stdcall

.data

.code
start:

end start
```

С помощью директивы .386P мы задаем модель процессора, команды которого будем использовать — в нашем случае все команды 80386. Команд процессора 80386 обычно достаточно для программирования большинства приложений под Windows. Если вам необходимы более современные команды, то вы можете задать более современную модель процессора, например .686P. Руководствуйтесь в выборе директивы задания набора допустимых команд процессора, пунктом 2.4.4.

Как и в DOS-программах следующей строкой мы устанавливаем модель памяти. Как уже говорилось, под Windows используется только плоская модель памяти (flat). Параметр stdcall необходим для того чтобы стало возможно вызывать функции API из программы на ассемблере. Мы еще поговорим об этом параметре.

Далее вы видите идут директивы задания сегментов данных (.data) и кода (.code). Как мы уже говорили в Windows нет сегментов по 64 Кбайт, однако директивы задания сегментов необходимы, чтобы просто указать ассемблеру, где в программе расположены данные, а где код. По этой причине часто вместо названия "сегмент" при программировании на ассемблере под Windows применяют название "секция", – мы тоже будем использовать это название.

Вы можете использовать в программах под Windows те же самые упрощенные директивы определения сегментов (.data, .data?, .const и пр.), что и в программах под DOS. Однако директива определения сегмента стека (.stack) обычно в программах под Windows не используется.

Выполнение кода начинается непосредственно после метки (я использую имя start, но вы можете выбрать для метки любое другое имя) и выполняется инструкция за инструкцией, если не встречаются команды перехода, такие как JMP, JNE, JE и т. п. Код программы завершается директивой END с указанием той же самой метки с которой начиналось выполнение, т. е. здесь все также как в DOS-программах.

Теперь в описанный каркас в секцию кода (.code) вы можете добавлять вызовы функций API, а в секцию данных (.data) все необходимые данные для вызова этих функций.

Как уже было сказано, программирование под Windows строится в основном на API-функциях, поэтому вам нужно знать какую функцию API нужно применить для решения конкретной задачи (также как при программировании под MS-DOS нужно было знать какую функцию DOS или BIOS использовать), а для этого вам нужно иметь список описания всех API-функций.

Мы в этой книге рассмотрим только самые основные функции API, т. к. описать все функции API в одной книге нереально. К тому же эта книга посвящена ассемблеру, а не программированию под Windows, поэтому я дам только общие концепции, покажу, как применять API в программах на ассемблере, но этого вам должно быть достаточно, чтобы вы самостоятельно начали писать программы любой сложности на ассемблере под Windows.

Полное описание всех функций API вы сможете найти в документации от Microsoft, называемой MSDN Library, которая распространяется в составе Visual Studio, а также бесплатно доступна в онлайн всем желающим по адресу: **http://msdn.microsoft.com/library/**. Обычным просмотром разделов этой

документации или, воспользовавшись строкой поиска по ключевым словам, вы сможете найти любую нужную API-функцию. Кроме описания API-функций в MSDN Library содержатся сведения о других языках Microsoft, например Visual Basic и Visual C#, в том числе есть сведения и по ассемблеру MASM, а также различные статьи по программированию. По этой причине найти что-либо нужное в MSDN Library не так просто, к тому же вся информация представлена только на английском языке. Поэтому вы можете поискать в интернете небольшие электронные справочники созданные русскими энтузиастами, в которых описаны некоторые основные API-функции на русском языке. Или еще лучше, вы можете купить какую-нибудь книгу-справочник по функциям Win 32 API (например [4], [5]). Ну и, разумеется, вы всегда можете на каком-нибудь форуме для программистов задать вопрос о выборе нужной функции API.

Прежде чем перейти к написанию программ рассмотрим общие концепции использования функций API в программах на ассемблере.

Все функции API имеют следующий общий вид согласно синтаксису языка Си (почти в таком виде они представлены в MSDN Library, только еще с указанием типов параметров и возвращаемого значения, о них будет рассказано ниже):

```
SomeFunc(a, b, c, d, e);
```

Это означает, что функция с именем `SomeFunc` должна получить пять параметров, с которыми она осуществит некоторые действия. Какие действия будет осуществлять функция, зависит от ее назначения. Параметрами обычно являются различные числовые значения или адреса в памяти, например, адрес по которому хранится строка для вывода в окне.

Существуют функции, в которые не нужно передавать никаких параметров, такие функции имеют следующий вид на языке Си:

```
SomeFunc();
```

Функции являются прямым аналогом процедур на ассемблере (перечитайте еще раз раздел 3.5). Следовательно, вызываются функции API в программах на ассемблере точно также как и процедуры, т. е. таким образом:

```
call SomeFunc
```

Однако перед вызовом функции ей нужно передать все параметры, в том случае, если она их имеет. Параметры для любой из функций API передаются только через стек, причем важен порядок передачи параметров — параметры помещаются в стек справа налево. Следовательно, вызов функции SomeFunc(a, b, c, d, e); на ассемблере будет записан следующим образом:

```
push e  ; первым заносим в стек самый правый параметр
push d
push c
push b
push a
call SomeFunc
```

Функции API самостоятельно очищают стек от переданных параметров по окончании своей работы, поэтому вам не нужно беспокоиться об этом.

Следует заметить, что если бы мы вызывали функции, написанные на языке Pascal или Basic, то параметры нужно было помещать в стек наоборот — слева направо.

Способ вызова API-функций на самом деле отличается от вызова стандартных функций языка Си, таких как printf, scanf и т. п. Функции API хотя и описываются в MSDN согласно синтаксису языка Си, но не являются Си-функциями. Согласно

конвенции Си параметры помещаются в стек справа налево также как и в вызове API-функций, но программист должен сам беспокоиться об освобождении стека. В этом случае для освобождения стека обычно используют команду `add ESP,n`, где `n` количество удаляемых байтов.

Именно поэтому в директиве `.model` мы указали параметр `stdcall`, т. к. он предусмотрен для функций API. Если бы мы вызывали функции языка Си, то в директиве `.model` нужно было бы указать параметр "C", а для вызова функций языка Pascal нужно было бы указать параметр "PASCAL". Смотрите еще раз описание директивы `.model` (разд. 2.3).

Многие функции API возвращают результат своего выполнения, например значение, сигнализирующее об успешном или неуспешном выполнении. Все API-функции, которые возвращают значение, записывают возвращаемое значение в регистр EAX.

Обычно в описании функции перед ее именем указывается тип возвращаемого значения, например:

```
int SomeFunc(a, b, c);
```

здесь int означает, что функция возвращает 32-разрядное значение со знаком.

Вот некоторые наиболее часто используемые типы (полный список можно найти в MSDN):

- int — 32-разрядное целое число со знаком;
- UINT — 32-разрядное беззнаковое целое число;
- BOOL — 32-разрядное целое, которое может принимать только два значения либо 0 (FALSE – ложь), либо 1 (TRUE – истина);
- LPTSTR — указатель на строку из 8- или 16-разрядных символов;
- LPCTSTR — тоже, что и LPTSTR, но используется для указания константных строк;
- WORD — 16-разрядное беззнаковое целое число;
- DWORD — 32-разрядное беззнаковое целое число;
- LONG — 32-разрядное целое число со знаком;
- ULONG — 32-разрядное беззнаковое целое число;
- VOID — функция не возвращает значение.

Эти же типы используются и в описании параметров передаваемых в функцию. Например:

```
VOID SomeFunc(LPTSTR a, DWORD b, LONG c);
```

Это запись означает, что в функцию нужно передать указатель на строку (параметр a), 32-разрядное беззнаковое целое число (параметр b) и 32-разрядное целое число со знаком (параметр c). Функция не возвращает никакого значения (тип VOID).

Еще одну важную концепцию, которую вам нужно знать при программировании под Windows, это то, что почти все объекты в Windows, такие как окна, кнопки, иконки, курсоры имеют так называемый хендл (handle) или по-русски — дескриптор. Дескриптор это просто уникальный 32-битный беззнаковый номер, который объект имеет в системе. Дескрипторы объектам присваивает сама ОС Windows.

Типы дескрипторов начинаются с буквы "H", например:

HWND — дескриптор окна;

HICON — дескриптор иконки;

HDC — дескриптор контекста устройства;

HANDLE — 32-разрядное целое число, используемое в качестве дескриптора.

В разд. 2.1 мы говорили о том, что при изучении программирования под ОС Windows будем использовать MASM32. Ассемблер MASM32 создан на основе MASM программистом-энтузиастом Стивом Хатчессоном (Steve Hutchesson). В наше время

программисты на ассемблере под Windows в основном используют только MASM32. Поэтому для продолжения вам необходимо скачать с сайта **http://www.masm32.com** и установить к себе на компьютер этот ассемблер.

Кроме документации, примеров программ, полезных утилит и пр. пакет MASM32 предоставляет специальные программные функции, которые через особые библиотеки входящие в пакет MASM32 программист может задействовать в своей программе. Эти функции имеют синтаксис подобный API-функциям Windows, но не являются API функциями. Следовательно, описание функций MASM32 невозможно найти в MSDN. Но их описание на английском языке можно найти в chm-файлах, которые входят в состав пакета MASM32, и расположены по следующему относительному пути: `\masm32\help\`.

В качестве примера в листинге 7.3 мы используем MASM32-функцию `FpuFLtoA`, которая предназначена для вывода вещественных чисел на экран. Эта функция не является API-функцией.

7.2. Первая простейшая программа под Windows на ассемблере

А теперь посмотрим, как написать простейшее приложение под Windows и на его примере рассмотрим все принципы вызовов API-функций на ассемблере. Все, что будет делать наша программа это выводить окно с сообщением, показанное на рис. 7.1.

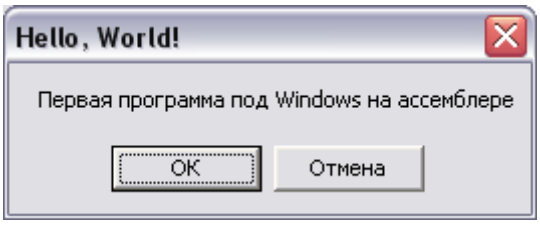

Рис. 7.1. Вид окна с сообщением, отображаемого программой

Для вывода таких сообщений используется API-функция MessageBox, ниже приведено описание, взятое из MSDN в переводе на русский:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Параметры функции:

- hWnd – дескриптор родительского окна. Если родительского окна нет, то используется нулевое значение;
- lpText – текст, отображаемый внутри окна;
- lpCaption – текст в заголовке окна;
- uType – тип окна сообщений, который позволяет задать, какие кнопки и иконки будут отображаться в окне. Этот параметр может быть комбинацией флагов из следующих групп флагов:
  - MB_ABORTRETRYIGNORE — окно сообщений будет содержать три кнопки: "Abort", "Retry", и "Ignore";
  - MB_OK — окно сообщений будет содержать только одну кнопку: "OK";
  - MB_OKCANCEL — окно сообщений будет содержать две кнопки: "OK" и "Cancel";
  - MB_YESNO — окно сообщений будет содержать две кнопки: "Yes" и "No";
  - MB_ICONEXCLAMATION — в окне сообщений появится иконка с восклицательным знаком;
  - MB_ICONHAND — в окне сообщений появится иконка с изображением знака "Stop";

- MB_ICONQUESTION — в окне сообщений появится иконка с изображением вопросительного знака;
- MB_ICONASTERISK — в окне сообщений появится иконка с изображением буквы "i";
- MB_DEFBUTTON1 — фокус находится на первой кнопке;
- MB_DEFBUTTON2 — фокус находится на второй кнопке;
- MB_DEFBUTTON3 — фокус находится на третьей кнопке.

Все флаги и вообще любые строковые идентификаторы, которые вы можете встретить в описаниях функций API, на самом деле являются именами определенных числовых значений (кодов). Вы не можете в программе на ассемблере просто использовать, скажем, флаг MB_OK, т. к. на самом деле требуется указать число, которое заменяет этот флаг. Узнать числовые значения флагов и всех прочих идентификаторов API-функций, всегда можно в так называемых заголовочных файлах (имеют расширение .h). В описании каждой API-функции в MSDN обычно указывается, в каком заголовочном файле следует искать такие числовые значения и сразу предоставляется возможность просмотреть этот заголовочный файл. Для функции MessageBox все значения флагов содержатся в заголовочном файле winuser.h. Вот нужный нам отрывок из winuser.h:

```
#define MB_OK                       0x00000000L
#define MB_OKCANCEL                 0x00000001L
#define MB_ABORTRETRYIGNORE         0x00000002L
#define MB_YESNOCANCEL              0x00000003L
#define MB_YESNO                    0x00000004L
#define MB_RETRYCANCEL              0x00000005L
```

Оператор языка Си #define является аналогом ассемблерной директивы эквивалентности EQU, которая присваивает имени значение операнда (см. разд. 2.4.2).

Как видим, нужному нам флагу MB_OKCANCEL присвоено значение 0x00000001L.

Буква L на конце числа означает в языке Си длинное целое (long). В ассемблере эта буква не используется, поэтому значение просто будет выглядеть как 1.

Мы можем в программе на ассемблере просто использовать это значение или с помощью директивы эквивалентности определить флаг:

```
MB_OKCANCEL EQU 1
```

К счастью нам не придется это делать самим постоянно, т. к. в пакет MASM32 входит специальный файл WINDOWS.INC (расположен в \MASM32\include\) в котором определены директивы эквивалентности для всевозможных идентификаторов и флагов, которые используют API-функции, в том числе и MB_OKCANCEL. Откройте в любом текстовом редакторе этот файл и посмотрите его содержимое. Можно (и нужно) подключать файл WINDOWS.INC в своих программах и использовать любые идентификаторы из него, как это сделать мы узнаем ниже.

Прежде чем мы начнем составлять программу на ассемблере, вам нужно узнать еще одну важную тонкость, связанную с API-функциями. Все API-функции работающие со строками (как, например, MessageBox) существуют в двух версиях:

- ANSI-версия, для работы со строками в кодировке ANSI (один байт на символ). В этом случае к имени функции добавляется суффикс A. Например, для функции MessageBox имя ANSI-версии будет MessageBoxA.
- Unicode-версия, для работы со строками в кодировке Unicode (два байта на символ). К имени функции добавляется суффикс W. Например, для функции MessageBox имя Unicode-версии будет MessageBoxW.

API-функции без суффиксов (например MessageBox), на самом деле является лишь обертками для этих двух функций. Компилятор высокоуровневых языков сам решает,

какую из двух версий API-функции нужно использовать, однако программист на ассемблере должен точно указывать нужную версию функции ANSI или Unicode в своих программах.

ANSI-версию стоит выбирать когда в функцию передаются строки только из 256 символов кодировки ASCII, а Unicode-версию, когда в функцию требуется передавать строки из расширенного набора (65536 символов) кодировки Unicode.

Строки для API-функций ANSI-версий (MessageBoxA) задаются также, как мы это делали в DOS-программах, к примеру:

```
str DB "Ivan Sklyaroff", 0
```

А для Unicode-версий (MessageBoxW) строки необходимо определять следующим образом:

```
str DW 'I', 'v', 'a', 'n', ' ', 'S', 'k', 'l', 'y', 'a', 'r', 'o', 'f',
'f', 0
```

Разумеется, можно просто указывать коды непечатаемых символов, например, в следующей строке заданы четыре символа в кодировке Unicode в шестнадцатеричном виде (значок "евро" – код 020ACh, математическая "бесконечность" – код 0221Eh, "больше или равно" – код 02265h, буква арабского языка – код 0642h):

```
str DW 020ACh, 0221Eh, 02265h, 0642h, 0
```

Мы далее будем использовать строки, а, следовательно, и функции только ANSI-версий.

Обратите также внимание, что в Windows строки заканчиваются нулем, а не символом $ как в DOS-программах. Это правило вытекает из языка Си, так как в нем строки должны заканчиваться нулевым символом. Однако помните о том, что часто один из параметров API-функции задает длину строки (в документации это часто называется буфером), определяемой другим параметром, из-за этого возвращаемая из функции строка может не иметь нуля на конце. При использовании в дальнейшем строки без нуля на конце, в других функциях могут возникнуть ошибки.

И еще одна важная деталь. В начале программы на ассемблере, также как и в программах на Си, нужно задавать так называемые прототипы всех вызываемых функций. Прототип это просто краткое описание функции с указанием точного числа параметров и их типов. Вызываемая функция должна соответствовать своему прототипу, иначе линкер выдаст ошибку. В языке программирования Си использование прототипов обычная практика, они помогают писать безошибочные программы.

Прототип создается с помощью директивы PROTO. Например, для функции MessageBoxA прототип будет выглядеть следующим образом:

```
MessageBoxA PROTO :DWORD,:DWORD,:DWORD,:DWORD
```

Это означает, что в функцию MessageBoxA должны быть переданы четыре 32-разрядных параметра.

В прототипах всех API-функций для указания типа параметров можно использовать DWORD, т. к. в API-функциях все параметры только 32-разрядные (мы говорили об этом выше).

Теперь мы можем написать предварительный вызов функции MessageBoxA на ассемблере:

```
MB_OKCANCEL EQU 1

.386P
```

```
.model flat,stdcall
MessageBoxA PROTO :DWORD,:DWORD,:DWORD,:DWORD

.data
hello_mess    db "Первая программа под Windows на ассемблере", 0
hello_title   db "Hello, World!", 0

.code
start:
       push MB_OKCANCEL
       push offset hello_title
       push offset hello_mess
       push 0
       call MessageBoxA

end start
```

Мы верно составили вызов API-функции, однако как мы говорили в начале этого дня, функции API находятся в dll-библиотеках, и пока не присоединить нужные библиотеки к программе никакие даже правильно составленные вызовы API-функций работать не будут. Но как присоединить динамические библиотеки к программе на ассемблере?

Напрямую это сделать нельзя и нужно использовать так называемые библиотеки импорта, которые обычно имеют тоже имя, что и соответствующие dll-файлы, но с расширением .lib. В MSDN Library в описании функции MessageBox можно увидеть следующую информацию (табл. 7.1).

Как видите, здесь указано, что функция содержится в файле user32.dll, а библиотекой импорта является файл User32.lib.

Таблица 7.1. *Информация из MSDN Library к описанию функции MessageBox*

Minimum DLL Version	user32.dll
Header	Declared in Winuser.h, include Windows.h
Import library	User32.lib
Minimum operating systems	Windows 95, Windows NT 3.1
Unicode	Implemented as ANSI and Unicode versions.

Файлы библиотек импорта входят в дистрибутивы любых средств разработки для Windows, например их можно взять из Visual Studio. Однако пакет MASM32 облегчает нам задачу тем, что в нем уже содержатся все необходимые библиотеки импорта в разделе \MASM32\lib\.

Библиотеки импорта подключается в программе на ассемблере с помощью директивы INCLUDELIB, следующим образом:

```
includelib \masm32\lib\user32.lib
```

Кроме того, в пакет MASM32 входят INC-файлы (расположены в разделе \MASM32\include\) в которых содержатся все прототипы API-функций. Эти INC-файлы имеют имена такие же, как имена соответствующих lib-файлов. Поэтому, подключив к программе INC-файл, вам не придется самостоятельно определять прототипы функций.

INC-файлы подключаются с помощью директивы INCLUDE, следующим образом:

```
include \masm32\include\user32.inc
```

Посмотрите содержимое файла user32.inc, вы найдете в нем прототип MessageBoxA.

Кроме того, как мы уже говорили, в разделе MASM32\include\ существует файл WINDOWS.INC, в котором определены директивы эквивалентности для

всевозможных идентификаторов и флагов, которые используют API-функции, как, например MB_OKCANCEL.

Подключается файл WINDOWS.INC также как и любой INC-файл:

```
include \masm32\include\windows.inc
```

Файл WINDOWS.INC является большим подспорьем для программистов, т. к. в больших программах число всевозможных флагов, идентификаторов и структур, необходимых функциям API достигает многие десятки. Подключив этот файл к своей программе, вам не придется больше определять их самостоятельно.

Мы далее будем включать этот файл во все наши программы на ассемблере.

В листинге 7.1 показана готовая программа, которая выводи окно сообщения.

Вы видите, мы задействовали еще одну API-функцию `ExitProcess` для выхода из программы. Описание функции:

```
VOID WINAPI ExitProcess(UINT uExitCode);
```

Она расположена в kernel32.dll, поэтому мы добавляем в программу включения файлов:

```
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
```

В качестве параметра uExitCode мы должны передать нулевое значение, чтобы завершить текущий процесс. Данная функция не возвращает никакого значения.

Функцией ExitProcess мы будем завершать все наши программы на ассемблере под Windows.

Компиляция программы осуществляется следующей строкой:

```
ml /c /coff /Cp hello.asm
```

Назначение параметров:

`/c` компиляция без компоновки.

`/coff` создать .obj файл в формате COFF (Common Object File Format). Применение этого параметра обязательно.

`/Cp` сохранять регистр имен, заданных пользователем. Вы можете вставить строку "option casemap:none" в начале исходного кода вашей программы, сразу после директивы .model, чтобы добиться того же эффекта.

После успешной компиляции `hello.asm`, вы получите `hello.obj`. Это объектный файл, который содержит инструкции и данные в двоичной форме. Отсутствуют только необходимая корректировка адресов, которая проводится линкером.

```
link.exe /SUBSYSTEM:WINDOWS /LIBPATH:c:\masm32\lib hello.obj
```

`/SUBSYSTEM:WINDOWS` информирует линкер о том, что приложение является оконным приложением. Для консольных приложений необходимо использовать параметр /SUBSYSTEM:CONSOLE.

`/LIBPATH:<путь к библиотекам импорта>` указывает линкеру местоположение библиотек импорта. Если вы используете MASM32, они будут в \MASM32\lib\.

После окончания линковки вы получите файл hello.exe.

Листинг 7.1. Простейшая программа под Windows (`hello.asm`)

```
        .386P
        .model flat,stdcall

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc

includelib \masm32\lib\user32.lib     ; Подключаем библиотеки
includelib \masm32\lib\kernel32.lib
```

```
        .data
hello_mess     db "Первая программа под Windows на ассемблере", 0
hello_title    db "Hello, World!", 0

        .code
start:
        push    MB_OKCANCEL
        push    offset hello_title
        push    offset hello_mess
        push    0
        call    MessageBoxA

        push    0
        call    ExitProcess

        end     start
```

7.2.1. Директива INVOKE

Вы можете для вызова API-функций использовать директиву INVOKE, которая позволяет записывать вызовы в более удобном виде. Например, следующий вызов:

```
push MB_OKCANCEL
push offset hello_title
push offset hello_mess
push 0
call MessageBoxA
```

с помощью директивы INVOKE будет записан следующим образом:

```
invoke MessageBoxA,0,addr hello_mess,addr hello_title, MB_OKCANCEL
```

Вызов функции ExitProcess можно записать следующим образом:

```
invoke ExitProcess, 0
```

В директиве INVOKE можно использовать оператор ADDR вместо OFFSET.

Далее в примерах кода мы будем использовать директиву INVOKE.

7.3. Консольное приложение

Хотя Windows является графической оконной системой, но вместе с тем она позволяет создавать консольные приложения, т. е. приложения которые подобно DOS-приложениям работают с командной строкой. Для создания консольных приложений в Windows существуют специальные API-функции.

Ввод и вывод в консоли выполняется с помощью функций ReadConsole и WriteConsole соответственно:

```
BOOL WINAPI WriteConsole(
  HANDLE hConsoleOutput,
  const VOID* lpBuffer,
  DWORD nNumberOfCharsToWrite,
  LPDWORD lpNumberOfCharsWritten,
  LPVOID lpReserved
);
```

Параметры функции:

- hConsoleOutput — дескриптор стандартного потока вывода;

- lpBuffer — указатель на строку которую нужно вывести;
- nNumberOfCharsToWrite — число символов строки, которые нужно вывести;
- lpNumberOfCharsWritten — указатель на переменную DWORD, куда будет помещено количество действительно выведенных символов после выполнения функции;
- lpReserved — зарезервированный параметр для использования в будущем, поэтому должен быть равен нулю.

```
BOOL WINAPI ReadConsole(
  HANDLE hConsoleInput,
  LPVOID lpBuffer,
  DWORD nNumberOfCharsToRead,
  LPDWORD lpNumberOfCharsRead,
  LPVOID lpReserved
);
```

Параметры функции:

- hConsoleInput — дескриптор стандартного потока ввода;
- lpBuffer — указатель на буфер, в который будет записана принятая строка;
- nNumberOfCharsToRead — максимальное число символов, которые будут приняты;
- lpNumberOfCharsRead — число фактически считанных символов;
- lpReserved — зарезервированный параметр для использования в будущем, поэтому должен быть равен нулю.

Дескрипторы ввода-вывода необходимо получить с помощью функции GetStdHandle:

```
HANDLE WINAPI GetStdHandle(DWORD nStdHandle);
```

Параметр nStdHandle может иметь одно из трех значений:

```
STD_INPUT_HANDLE      equ -10
STD_OUTPUT_HANDLE     equ -11
STD_ERROR_HANDLE      equ -12
```

Эти идентификаторы определены в файле WINDOWS.INC.

В листинге 7.2 показана простейшая программа, которая просит пользователя ввести свое имя и затем приветствует его по имени, например:

```
Your name: Ivan Sklyaroff
Hello, Ivan Sklyaroff
```

С помощью API-функции

```
BOOL WINAPI SetConsoleTitle(LPCTSTR lpConsoleTitle);
```

мы выводим в заголовке консоли фразу "DEMO CONSOLE PROGRAM".

Единственный аргумент этой функции — адрес строки символов, которая будет выведена в заголовке окна консоли.

Компиляция:

```
ml /c /coff /Cp conwin.asm
link.exe /SUBSYSTEM:CONSOLE /LIBPATH:d:\masm32\lib conwin.obj
```

Листинг 7.2. Консольная программа под Windows (`conwin.asm`)

```
        .386
        .model flat,stdcall

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
```

```
includelib \masm32\lib\kernel32.lib

        .const
ConsoleTitle   db     "DEMO CONSOLE PROGRAM",0

        .data
msg1    db     "Your name: "
msg1_len = $-msg1
msg2    db     "Hello, "
msg2_len = $-msg2
InputBuffer    db     100 dup (?)    ; буфер для ввода
len_InputBuffer = $-InputBuffer

        .data?
hOutputdd      ?       ; хендл для вывода
hInput  dd     ?       ; хендл для ввода
nRead   dd     ?       ; прочитано байт
nWritendd      ?       ; напечатано байт

        .code
start:
        invoke SetConsoleTitleA,addr ConsoleTitle

        ; получаем хендл для вывода
        invoke GetStdHandle,STD_OUTPUT_HANDLE
        mov    hOutput,eax

        invoke WriteConsoleA,hOutput,addr msg1,msg1_len,addr nWriten,NULL

        ; получаем хендл для ввода
        invoke GetStdHandle,STD_INPUT_HANDLE
        mov    hInput,eax

        invoke ReadConsoleA,hInput,addr InputBuffer,len_InputBuffer,addr
nRead,NULL
        invoke WriteConsoleA,hOutput,addr msg2,msg2_len,addr nWriten,NULL
        invoke WriteConsoleA,hOutput,addr InputBuffer,nRead,addr
nWriten,NULL
        invoke ExitProcess,0

        end    start
```

В листинге 7.3 показано еще одно консольное приложение. Данная Windows-программа является аналогом MS-DOS программы из листинга 5.2, которая складывает два вещественных числа с плавающей запятой.

Кроме функции FpuFLtoA все остальные функции в этой программе вам уже знакомы. Функция FpuFLtoA предназначена для вывода вещественного числа (в нашем случае результат сложения вещественных чисел) на экран. Данная функция не является API-функцией, а является внутренней MASM32-функцией и входит в специальную библиотеку FPU.LIB, которая подключается в начале нашей программы также как и остальные библиотеки. Подробное описание функции на английском языке можно найти в справке MASM32, которая обычно располагается в каталоге \masm32\help\fpuhelp.chm.

Функция FpuFLtoA может работать только с 80-битовыми числами и имеет четыре параметра:

```
FpuFLtoA (
        lpSrc1
        lpSrc2
        lpszDest
        uID
)
```

- lpSrc1 — адрес 80-битового числа, которое будет выводиться на экран (этот параметр игнорируется, если uID имеет значение SRC1_FPU);
- lpSrc2 — беззнаковое целое указывающее количество десятичных знаков после запятой или адрес беззнакового целого (в зависимости от параметра uID);
- lpszDest — адрес буфера, куда будут записаны символы, в которые преобразуется число. Буфер должен быть достаточно большим, чтобы вместить возвращаемые символы. Самое большое число может иметь 17 символов перед знаком десятичной точки, знак десятичной точки, 15 десятичных цифр после точки, и завершающий нуль (таким образом, максимум 34 символа);
- uID — флаги, управляющие работой функции. Один из флагов SRC1_? может объединяться с помощью операции OR только с одним из SRC2_? и одним из STR_? флагов (Флаг STR_REG не нужно объединять оператором OR, если строка должна быть возвращена в десятичном формате; потому что по умолчанию так).

uID флаги и их назначение:

- SRC1_FPU — первый параметр функции Src не используется (уже на FPU);
- SRC1_REAL — первый параметр Src должен быть адресом 80-битного числа хранящегося в обычной памяти;
- SRC2_DMEM — второй параметр Src2 должен быть адресом 32-битного беззнакового числа;
- SRC2_DIMM — второй параметр Src2 должен быть простым 32-битным беззнаковым числом;
- STR_REG — строка должна быть возвращена в десятичном формате;
- STR_SCI — строка должна быть возвращена в научном формате.

Компиляция:

```
ml /c /coff /Cp fpuconwin.asm
link.exe /SUBSYSTEM:CONSOLE /LIBPATH:d:\masm32\lib fpuconwin.obj
```

Листинг 7.3. Консольная программа под Windows складывающая два вещественных числа (`fpuconwin.asm`)

```
        .386
        .model flat,stdcall
        option  casemap:none

include     \masm32\include\windows.inc
include     \masm32\include\kernel32.inc
include     \masm32\include\fpu.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\fpu.lib

BSIZE   equ     30

        .data
num1    dd      45.56   ; первое вещественное число
num2    dd      30.13   ; второе вещественное число
```

```
result dt     ?      ; сюда будет помещен результат
stdout dd     ?      ; хендл для вывода
cWritten  dd  ?      ; хендл для ввода
buf       db  BSIZE dup (?)

       .code
start:
       main   proc
       invoke  GetStdHandle,STD_OUTPUT_HANDLE
       mov    stdout,eax
       fld    num1   ; загружаем 1-й операнд в ST
       fld    num2   ; 2-й операнд в ST, 1-й операнд в ST(1)
       fadd          ; складываем: сумма помещается в ST
       fstp   result ; скопировать из ST в result
       invoke  FpuFLtoA,ADDR result,5,ADDR buf,SRC1_REAL or SRC2_DIMM
       invoke  WriteConsoleA,stdout,ADDR buf,BSIZE,ADDR cWritten,NULL
       invoke ExitProcess,0
       main   endp

       end    start
```

7.4. Графическое приложение

В предыдущем разделе мы увидели, как просто создаются консольные приложения на ассемблере под Windows. Однако основными приложениями в Windows являются графические оконные приложения. Мы уже рассматривали создание простого окна сообщения с помощью единственной API-функции MessageBox. Однако построение стандартных окон приложений Windows, таких как на рис. 7.2, намного сложнее и одной API-функцией уже не обойтись.

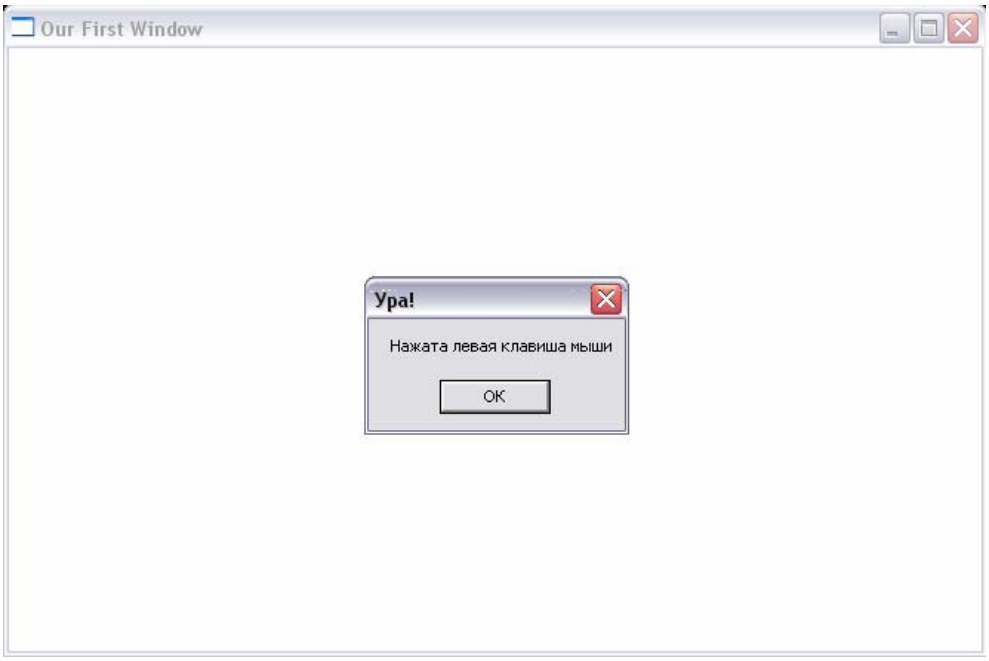

Рис. 7.2. Простейшее оконное графическое приложение

Прежде чем перейти к программированию стандартного окна, необходимо понять принципы работы графических приложений в Windows.

Функционирование ОС Windows основано на сообщениях. Сообщение — это небольшой пакет данных, который генерируется в результате определенного произошедшего события в системе. Например, после нажатия пользователем клавиши на клавиатуре, щелчка или перемещения указателя мыши, изменения размера окна, выбора пункта меню, перемещении бегунка прокрутки и т. д. Сообщения могут генерироваться системным таймером и самой Windows, например, в случае завершения работы системы Windows посылает каждому открытому окну сообщение о необходимости проверить сохранность данных и закрыть окна. Существуют тысячи различных видов сообщений, но все они представлены следующей структурой (определена в winuser.h):

```
typedef struct tagMSG {     // msg
   HWND hwnd;
   UINT message;
   WPARAM wParam;
   LPARAM lParam;
   DWORD time;
   POINT pt;
} MSG;
```

Параметры:

- hwnd — дескриптор окна, которому адресуется сообщение;
- message — тип Windows-сообщения (например WM_RBUTTONDOWN – нажатие правой кнопки мыши). Имена всех типов сообщений начинаются с приставки WM_;
- wParam — параметр сообщения, содержащий дополнительные сведения о сообщении, например координаты указателя мыши;
- lParam — параметр сообщения, содержащий дополнительные сведения о сообщении;
- time — время постановки сообщения в очередь;
- pt — координаты мыши в момент time.

На ассемблере эта структура с помощью неизученной пока нами директивы STRUC будет выглядеть следующим образом:

```
MSG STRUCT
  hwnd      DWORD     ?
  message   DWORD     ?
  wParam    DWORD     ?
  lParam    DWORD     ?
  time      DWORD     ?
  pt        DWORD     ?
MSG ENDS
```

В таком виде эта структура определена в файле WINDOWS.INC. Чтобы использовать поля этой структуры в программе нужно с помощью директивы INCLUDE подключить файл WINDOWS.INC и в сегменте данных сделать следующее определение:

```
msg MSG <?>
```

Знак вопроса означает, что все поля структуры будут инициализированы неизвестным значением, но можно, например, инициализировать их нулем:

```
msg MSG <0>
```

Доступ к отдельным полям осуществляется с помощью точки:

```
mov msg.hwnd, EAX ; запись значения из EAX в поле hwnd
mov msg.lParam, EAX ; запись значения из EAX в поле lParam
```

Всего существует четыре основных источника, от которых приложение может получать сообщения: пользователь, ОС Windows, само приложение (от самого себя), другие приложения.

Все сообщения независимо от того, чем они сгенерированы и какому приложению предназначены, первоначально помещаются в так называемую системную очередь сообщений Windows. Windows выполняет обработку некоторых сообщений самостоятельно, однако большая их часть передается для обработки приложениям. Следовательно, для того чтобы приложение смогло обрабатывать сообщения в нем должна быть организована локальная очередь сообщений. Приложение должно периодически проверять свою локальную очередь на наличие новых сообщений и выполнять их обработку. Для этого в приложении необходимо организовать цикл и создать функцию для обработки поступающих сообщений.

Существует два типа сообщений: синхронные и асинхронные. Очередь синхронных сообщений формируется по принципу FIFO (first in, first out — первым прибыл, первым обслужен). Асинхронные сообщения Windows всегда вставляет сразу в голову очереди. Например, сообщения о перерисовке окна, сообщения таймера и сообщения о завершении программы являются асинхронными. Они имеют наивысший приоритет и всегда передаются раньше всех других сообщений.

Это справедливо как для системной очереди сообщений, так и для локальных очередей приложений, т. к. локальные очереди формируются от системной очереди.

Таким образом, полноценное графическое приложение будет состоять из следующих основных частей:

- регистрация класса окон;
- создание окна;
- цикл обработки очереди сообщений;
- процедура главного окна.

Рассмотрим каждую часть графического приложения, а полный исходный код показан в листинге 7.4.

7.4.1. Регистрация класса окон

Класс окна это комбинация свойств окна, дескрипторы значка и указателя мыши, а также прочие атрибуты, которые будут являться шаблоном для всех экземпляров окон приложения.

Класс регистрируется с помощью API-функции RegisterClass, которая имеет только один аргумент – указатель на структуру типа WNDCLASS. Структура WNDCLASS хранит характеристики создаваемого класса, поэтому перед регистрацией необходимо заполнить поля этой структуры.

```
typedef struct {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

Параметры:

- style — определяет свойства окна. Все идентификаторы свойств окна начинаются с приставки CS_ (ищите их полный перечень в MSDN). Можно задать комбинацию свойств;
- lpfnWndProc — указатель на оконную процедуру, которая будет обрабатывать сообщения, получаемые окном;

- cbClsExtra — количество байт, резервируемых в памяти после создания структуры класса окна. Может быть равным NULL;
- cbWndExtra — количество байт, резервируемых в памяти после создания экземпляра окна. Может быть равным NULL;
- hInstance — дескриптор экземпляра приложения, регистрирующего класс окна;
- hIcon — дескриптор иконки, которая будет символом окна данного класса. Все идентификаторы предопределенных иконок начинаются с приставки IDI_. Этот параметр может быть равным NULL;
- hCursor — дескриптор указателя мыши, используемого в окне приложения. Все идентификаторы предопределенных курсоров начинаются с приставки IDC_ (вероятно "IDentificator of Cursor"). Этот параметр может быть равным NULL;
- hbrBackground — дескриптор кисти (brush), которой будет закрашен фон окна;
- lpszMenuName — указатель на строку, завершающуюся нулевым символом и содержащую имя меню для данной программы. Если программа не работает с меню, то этот параметр может быть NULL;
- lpszClassName — указатель на строку, завершающуюся нулевым символом и содержащую имя создаваемого класса окна.

Существует расширенный вариант структуры WNDCLASS, получивший название WNDCLASSEX:

```
typedef struct {
    UINT cbSize;
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
    HICON hIconSm;
} WNDCLASSEX, *PWNDCLASSEX;
```

От WNDCLASS он отличается наличием двух дополнительных полей:

- cbSize — задает размер всей структуры;
- hIconSm — содержит дескриптор малого значка.

В файле WINDOWS.INC структура WNDCLASSEX определена следующим образом:

```
WNDCLASSEX STRUCT
  cbSize            DWORD     ?
  style             DWORD     ?
  lpfnWndProc       DWORD     ?
  cbClsExtra        DWORD     ?
  cbWndExtra        DWORD     ?
  hInstance         DWORD     ?
  hIcon             DWORD     ?
  hCursor           DWORD     ?
  hbrBackground     DWORD     ?
  lpszMenuName      DWORD     ?
  lpszClassName     DWORD     ?
  hIconSm           DWORD     ?
WNDCLASSEX ENDS
```

Чтобы использовать поля этой структуры в программе нужно с помощью директивы INCLUDE подключить файл WINDOWS.INC и в сегменте данных сделать следующее определение:

```
wc WNDCLASSEX <?>
```

Доступ к отдельным полям осуществляется с помощью точки:

```
mov wc.lpszMenuName,NULL ; запись нулевого значения в поле lpszMenuName
mov wc.lpszClassName,OFFSET ClassName ; запись адреса строки с именем
                                      ; создаваемого класса окна
                                      ; в поле lpszClassName
```

7.4.2. Создание окна

После регистрации класса окна можно создать окно вызовом API-функции CreateWindow. После успешного выполнения эта функция возвращает дескриптор созданного окна, в противном случае – NULL.

```
HWND CreateWindow(
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam
);
```

Параметры функции:

- lpClassName — указатель на строку с именем класса. В большинстве случаев значение этого параметра совпадает со значением последнего поля структуры WNDCLASS, передаваемой RegisterClass;
- lpWindowName — указатель на строку содержащую строку заголовка окна;
- dwStyle — стиль окна, который определяет, будет ли окно иметь заголовок, иконку системного меню, кнопки минимизации, максимизации, рамка окна и т. д. Идентификаторы стиля начинаются с приставки WS_;
- x — отступ левого верхнего угла окна от левого края экрана в пикселах;
- y — отступ левого верхнего угла окна от верхней границы экрана в пикселах;
- nWidth — ширина окна в пикселах;
- nHeight — высота окна в пикселах;
- hWndParent — дескриптор родительского окна;
- hMenu — дескриптор меню;
- hInstance — дескриптор экземпляра приложения;
- lpParam — дополнительные значения. Если окну не передаются никакие дополнительные значения, этот параметр должен быть равен NULL.

7.4.3. Цикл обработки очереди сообщений

На языке Си цикл обработки очереди сообщений выглядит следующим образом:

```
while (GetMessage(&msg, NULL, 0, 0))
{
        TranslateMessage(&msg);
```

```
        DispatchMessage(&msg);
}
```

Как видно в цикле используется три API-функции: GetMessage, TranslateMessage, DispatchMessage.

Функция GetMessage выбирает из очереди очередное сообщение и записывает его в структуру msg. Второй параметр этой функции — дескриптор окна, созданного программой. Если этот параметр равен NULL, то обрабатываться будут сообщения для всех созданных программой окон.

Третий и четвертый параметры позволяют задать фильтр, для того чтобы оконной процедуре передавались сообщения, номера которых попадают в определенный интервал. Третий параметр – нижняя граница этого интервала, четвертый – верхняя граница. Если эти параметры равны нулю, как в нашем случае, то принимаются все сообщения, адресованные данному приложению.

Цикл обработки сообщений останавливается, а, следовательно, прекращается работа программы, только после получения единственного сообщения WM_QUIT.

Функция TranslateMessage преобразует сообщения о нажатии виртуальных клавиш в символьные сообщения. Например, сообщения WM_KEYDOWN и WM_KEYUP в WM_CHAR и WM_DEADCHAR, а также WM_SYSKEYDOWN и WM_SYSKEYUP в WM_SYSCHAR и WM_SYSDEADCHAR. Эта функция необходима только тем приложениям, которые обрабатывают ввод с клавиатуры. Например, она позволяет пользователям выбирать команды меню нажатием клавиш, а не только щелчками мыши.

Функция DispatchMessage передает сообщение на обработку соответствующей оконной процедуре, автоматически определяя окно, которому адресовано сообщение.

На ассемблере цикл обработки сообщений может выглядеть следующим образом:

```
        MSG_LOOP:
                invoke GetMessage, ADDR msg,NULL,0,0

                cmp eax,0
                je END_LOOP
                invoke TranslateMessage, ADDR msg
                invoke DispatchMessage, ADDR msg
                jmp MSG_LOOP
        END_LOOP:
                mov     eax,msg.wParam
                invoke ExitProcess,eax
```

или более короткий вариант:

```
    .WHILE TRUE
       invoke GetMessage, ADDR msg,NULL,0,0
    .BREAK .IF (!eax)
       invoke TranslateMessage, ADDR msg
       invoke DispatchMessage, ADDR msg
    .ENDW
     mov     eax,msg.wParam ; сохранение возвращаемого значения в eax
       invoke ExitProcess,eax
```

7.4.4. Процедура главного окна

Прототип оконной процедуры на языке Си выглядит следующим образом:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
```

Параметры, передаваемые в оконную функцию: hWnd — дескриптор окна, которому посылается сообщение, message — идентификатор сообщения, wParam и lParam — дополнительная информация, зависящая от конкретного сообщения.

Все четыре параметра в ассемблере имеют тип DWORD.

Компиляция:

```
ml /c /coff /Cp gdiwin.asm
link.exe /SUBSYSTEM:WINDOWS /LIBPATH:d:\masm32\lib gdiwin.obj
```

Листинг 7.4. Простейшая графическая программа под Windows (gdiwin.asm)

```
        .386
        .model flat,stdcall
        option casemap:none

include\masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

        .data
ClassName       db "SimpleWinClass",0
AppName         db "Our First Window",0

; Для функции MessageBox
hello_mess1     db "Нажата левая клавиша мыши",0
hello_mess2     db "Нажата правая клавиша мыши",0
hello_title     db "Ура!",0

        .data?
hInstance       HINSTANCE ?
hInst           HINSTANCE ?
CommandLine     LPSTR ?
wc              WNDCLASSEX <?>
msg             MSG <?>
hwnd            HWND ?

        .code
start:
        invoke GetModuleHandle,NULL
        mov    hInstance,eax

        mov    wc.cbSize,SIZEOF WNDCLASSEX
        mov    wc.style,CS_HREDRAW or CS_VREDRAW
        mov    wc.lpfnWndProc,OFFSET WndProc
        mov    wc.cbClsExtra,NULL
        mov    wc.cbWndExtra,NULL
        push   hInstance
        pop    wc.hInstance
        mov    wc.hbrBackground,COLOR_WINDOW+1
        mov    wc.lpszMenuName,NULL
```

```
        mov     wc.lpszClassName,OFFSET ClassName
        invoke  LoadIcon,NULL,IDI_APPLICATION
        mov     wc.hIcon,eax
        mov     wc.hIconSm,eax
        invoke  LoadCursor,NULL,IDC_ARROW
        mov     wc.hCursor,eax
        invoke  RegisterClassEx,addr wc
        INVOKE  CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
            WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
            CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,\
            hInst,NULL
        mov     hwnd,eax
        invoke  ShowWindow,hwnd,SW_SHOWNORMAL
        invoke  UpdateWindow,hwnd

        MSG_LOOP:
                invoke  GetMessage,ADDR msg,NULL,0,0
                cmp     eax,0
                je      END_LOOP
                invoke  TranslateMessage,ADDR msg
                invoke  DispatchMessage,ADDR msg
                jmp     MSG_LOOP
        END_LOOP:
                mov      eax,msg.wParam
                invoke  ExitProcess,eax

WndProc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM

        cmp     uMsg,WM_DESTROY
        je      WMDESTROY
        cmp     uMsg,WM_LBUTTONDOWN
        je      LBUTTON
        cmp     uMsg,WM_RBUTTONDOWN
        je      RBUTTON
        JMP     DEFWNDPROC
WMDESTROY:
        invoke  PostQuitMessage,NULL
        xor     eax,eax
        jmp     FINISH
LBUTTON:
        invoke  MessageBoxA,0,addr hello_mess1,addr hello_title,MB_OK
        jmp     FINISH
RBUTTON:
        invoke  MessageBoxA,0,addr hello_mess2,addr hello_title,MB_OK
        jmp     FINISH

DEFWNDPROC:
        invoke  DefWindowProc,hWnd,uMsg,wParam,lParam
FINISH:
        ret
        WndProc endp
```

```
        end     start
```

7.5. Дочерние окна управления

Мы научились создавать простое оконное приложение. Теперь в окне можно размещать элементы управления – кнопки (buttons), флажки (check boxes), окна редактирования (edit boxes), списки (list boxes), комбинированные списки (combo boxes), строки текста (text strings), полосы прокрутки (scroll bars) и т. д.

Элементы управления на самом деле являются тоже окнами и поэтому их принято называть дочерними окнами управления (child window controls).

Дочернее окно, также как обычное окно, создается с помощью функции CreateWindow или CreateWindowEx. Но в случае стандартных элементов управления регистрировать класс окна функцией RegisterClassEx не нужно. Такой класс уже существует (зарегистрирован) в Windows и имеет одно из следующих имен: "button" (кнопка), "static" (статическое), "scrollbar" (полоса прокрутки), "edit" (окно редактирования), "listbox" (окно списка) или "combobox" (окно комбинированного списка). Вам необходимо только использовать одно из этих имен в качестве параметра класса окна в функции CreateWindow. Другие параметры, которые необходимо указать в этой функции - это дескриптор родительского окна и идентификатор (ID) элемента управления. ID элемента управления должен быть уникальным, он необходим для того, чтобы отличать элементы управления друг от друга.

Обычно дочерние окна создаются во время обработки сообщения WM_CREATE главного окна:

```
.ELSEIF uMsg==WM_CREATE
        invoke CreateWindowEx,WS_EX_CLIENTEDGE, ADDR EditClassName,NULL,\
             WS_CHILD or WS_VISIBLE or WS_BORDER or ES_LEFT or\
                   ES_AUTOHSCROLL,\
                   50,25,300,25,hWnd,EditID,hInstance,NULL
        mov  hwndEdit,eax
        invoke SetFocus, hwndEdit
        invoke CreateWindowEx,NULL, ADDR ButtonClassName,ADDR ButtonText,\
                   WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
                   50,60,100,25,hWnd,ButtonID,hInstance,NULL
        mov  hwndButton,eax
```

Вы видите, что в первом вызове CreateWindowEx создается окно редактирования ("edit"), а во втором вызове CreateWindowEx кнопка ("button"). Кроме обычных стилей окна каждый элемент управления имеет свои дополнительные стили. Например, стили кнопок имеют приставку "BS_", а стили окон редактирования - "ES_". Вам придется посмотреть информацию об этих стилях в MSDN или вашем справочнике по Win32 API, т. к. чтобы привести их все нужно отводить отдельную главу книги.

Функция SetFocus вызывается для того, чтобы установить фокус ввода на окно редактирования, для того чтобы пользователь мог сразу вводить в нем текст.

Обратите также внимание, что после создания каждого элемента управления, его дескриптор сохраняется в соответствующей переменной (hwndEdit и hwndButton) для будущего использования.

После того как элемент управления создан, он в случае изменения своего состояния (например, нажатия кнопки) посылает сообщение WM_COMMAND родительскому окну, при этом в младшем слове wParam передается ID элемента, в старшем слове wParam код события, а в lParam дескриптор элемента. В программе по значению этих параметров можно определить, с каким элементом и что произошло:

```
.ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
```

```
        .IF lParam==1
                .IF ax==IDM_GETTEXT
                invoke GetWindowText,hwndEdit,ADDR buffer,512
                invoke MessageBox,NULL,ADDR buffer,ADDR AppName,MB_OK
```

API-функция GetWindowText принимает текст из окна редактирования, который затем выводится на экран функцией MessageBox.

Участок кода, который обрабатывает нажатие на кнопку, показан ниже:

```
.IF ax==ButtonID
     shr eax,16
        .IF ax==BN_CLICKED
            invoke SendMessage,hWnd,WM_COMMAND,IDM_GETTEXT,1
        .ENDIF
.ENDIF
```

Сначала проверяется младшее слово параметра wParam, чтобы убедиться, что он содержит ID кнопки. Если это так, далее проверяется старшее слово wParam, чтобы убедиться, что был послан код уведомления BN_CLICKED, то есть кнопка была нажата.

Функция SendMessage посылает окну сообщение WM_COMMAND с параметрами lParam=1 и IDM_GETTEXT в старшем слове wParam.

Описание функции SendMessage:

```
LRESULT SendMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
```

Параметры:

- hWnd — дескриптор окна, которому посылается сообщение. Если этот параметр равен HWND_BROADCAST, то сообщение посылается всем всплывающим окнам в системе, включая невидимые и окна без фокуса ввода и т. д., но кроме дочерних окон;
- Msg — тип сообщения;
- wParam — дополнительная информация о сообщении;
- lParam — дополнительная информация о сообщении.

Компиляция программы:

```
ml /c /coff /Cp wincontrols.asm
link /SUBSYSTEM:WINDOWS /LIBPATH:d:\masm32\lib wincontrols.obj
```

На рис. 7.3 показан результат работы программы.

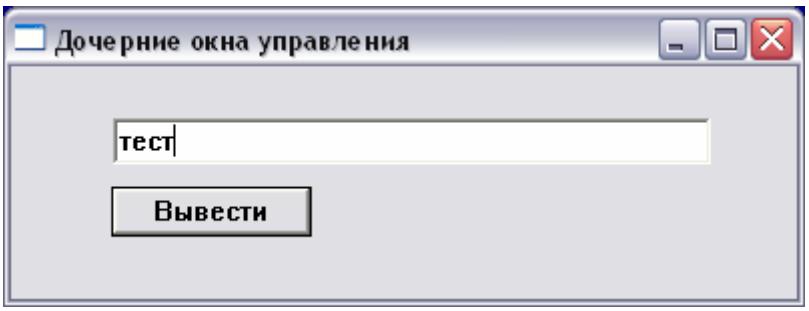

Рис. 7.3. Окно программы с элементами управления

Листинг 7.5. Пример окна с элементами управления (wincontrols.asm)

```
        .386
        .model flat,stdcall
        option casemap:none

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

        .data
ClassName      db     "SimpleWinClass",0
AppName        db     "Дочерние окна управления",0
ButtonClassName db    "button",0
ButtonText     db     "Вывести",0
EditClassName  db     "edit",0

        .data?
hInstance      HINSTANCE      ?
CommandLine    LPSTR          ?
hwndButton     HWND           ?
hwndEdit       HWND           ?
buffer         db             512 dup(?)

        .const
ButtonID       equ    1
EditID         equ    2
IDM_HELLO      equ    1
IDM_CLEAR      equ    2
IDM_GETTEXT    equ    3
IDM_EXIT       equ    4

        .code
start:
        invoke GetModuleHandle,NULL
        mov    hInstance,eax
        invoke GetCommandLine
        mov    CommandLine,eax
        invoke WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
        invoke ExitProcess,eax

WinMain proc
hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
        LOCAL  wc:WNDCLASSEX
        LOCAL  msg:MSG
        LOCAL  hwnd:HWND
        mov    wc.cbSize,SIZEOF WNDCLASSEX
        mov    wc.style, CS_HREDRAW or CS_VREDRAW
```

```
        mov     wc.lpfnWndProc, OFFSET WndProc
        mov     wc.cbClsExtra,NULL
        mov     wc.cbWndExtra,NULL
        push    hInst
        pop     wc.hInstance
        mov     wc.hbrBackground,COLOR_BTNFACE+1
        mov     wc.lpszMenuName,NULL
        mov     wc.lpszClassName,OFFSET ClassName
        invoke  LoadIcon,NULL,IDI_APPLICATION
        mov     wc.hIcon,eax
        mov     wc.hIconSm,eax
        invoke  LoadCursor,NULL,IDC_ARROW
        mov     wc.hCursor,eax
        invoke  RegisterClassEx, addr wc
        INVOKE  CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
            WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
            CW_USEDEFAULT,400,150,NULL,NULL,\
            hInst,NULL
        mov     hwnd,eax
        INVOKE  ShowWindow, hwnd,SW_SHOWNORMAL
        INVOKE  UpdateWindow, hwnd
        .WHILE  TRUE
                INVOKE GetMessage, ADDR msg,NULL,0,0
                  .BREAK .IF (!eax)
                  INVOKE TranslateMessage, ADDR msg
                  INVOKE DispatchMessage, ADDR msg
        .ENDW
                mov     eax,msg.wParam
        ret
WinMain endp

WndProc proc hWnd:HWND,uMsg:UINT,wParam:WPARAM,lParam:LPARAM
        .IF uMsg==WM_DESTROY
                invoke PostQuitMessage,NULL
        .ELSEIF uMsg==WM_CREATE
                invoke CreateWindowEx,WS_EX_CLIENTEDGE, ADDR
EditClassName,NULL,\
                          WS_CHILD or WS_VISIBLE or WS_BORDER or ES_LEFT
or\
                          ES_AUTOHSCROLL,\
                          50,25,300,25,hWnd,EditID,hInstance,NULL
                mov     hwndEdit,eax
                invoke  SetFocus, hwndEdit
                invoke  CreateWindowEx,NULL, ADDR ButtonClassName,ADDR
ButtonText,\
                          WS_CHILD or WS_VISIBLE or BS_DEFPUSHBUTTON,\
                          50,60,100,25,hWnd,ButtonID,hInstance,NULL
                mov     hwndButton,eax
        .ELSEIF uMsg==WM_COMMAND
                mov     eax,wParam
                .IF lParam==1
                        .IF ax==IDM_GETTEXT
```

```
                        invoke GetWindowText,hwndEdit,ADDR
buffer,512
                        invoke MessageBox,NULL,ADDR buffer,ADDR
AppName,MB_OK
                .ELSE
                        invoke DestroyWindow,hWnd
                .ENDIF
        .ELSE
                .IF ax==ButtonID
                        shr eax,16
                        .IF ax==BN_CLICKED
                                invoke
SendMessage,hWnd,WM_COMMAND,IDM_GETTEXT,1
                        .ENDIF
                .ENDIF
        .ENDIF
.ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
.ENDIF
xor     eax,eax
ret
WndProc endp

        end     start
```

7.6. Использование ресурсов

Важной концепцией программирования под Windows является понятие ресурсов (resource). Большинство графических программ под Windows не обходятся без ресурсов. Ресурсы это визуальные и логические элементы программ, которые хранятся в одном файле с программой, но отдельно от кода и данных. Особенностью ресурсов является то, что они загружаются в память только при обращении к ним, благодаря чему достигается экономия памяти. Это важно, т. к. многие ресурсы могут занимать десятки и сотни мегабайт.

Наиболее часто используемыми стандартными ресурсами являются:

- меню;
- диалоговые окна;
- строки символов;
- курсоры;
- пиктограммы (иконки);
- растровые изображения;
- шрифты.

Я перечислил далеко не все стандартные ресурсы, полный список вы можете увидеть в MSDN. Кроме стандартных ресурсов программист может создавать и использовать нестандартные ресурсы.

Отличие стандартных ресурсов от нестандартных в том, что для стандартных ресурсов имеются предопределенные Win API, которые позволяют программисту манипулировать стандартными ресурсами. В случае нестандартных ресурсов работа с ними ложится исключительно на программиста, Windows способна только загрузить в память нестандартный ресурс и предоставить на него указатель.

Мы рассмотрим работу только с некоторыми наиболее часто используемыми стандартными ресурсами.

Здесь же стоит заметить, что существуют функции API, которые могут заменить ресурсы.

Например, меню можно определять в ресурсах, а можно программно с помощью функций Win32 API. Возможен также комбинированный подход, когда часть меню создается в ресурсах, а другая часть с помощью API-функций.

7.6.1. Подключение ресурсов к исполняемому файлу

Для подключения ресурсов в программу нужно создать их описание на специальном скриптовом языке и сохранить это описание в текстовый файл с расширением .RC. С языком описания ресурсов мы познакомимся позже.

Затем полученный текстовый файл необходимо откомпилировать специальным компилятором ресурсов. В пакет MASM32 входит компилятор ресурсов RC.EXE. В результате получится объектный файл с расширением .RES.

В исполняемый файл RES-файл подключается компоновщиком.

Файлы RC и RES вы можете создавать с помощью специальных редакторов ресурсов, которые входят, например, в пакеты Visual Studio и Borland. Ресурсы с помощью этих редакторов создаются в визуальном режиме, поэтому от программиста не требуется знание языка описания ресурсов.

Если вам будет нужно, вы научитесь самостоятельно работать с этими программами, т. к. это очень просто, а мы в этой книге изучим язык описания ресурсов.

7.6.2. Язык описания ресурсов

7.6.2.1. Пиктограммы

Пиктограмма описывается одной строкой вида:

```
имя_ID ICON имя_файла
```

Где имя_ID — целочисленный идентификатор ресурса

имя_файла — имя файла иконки. Если файл находится не в текущей директории, то необходимо указывать полный путь к файлу.

Примеры описания иконок:

```
123 ICON "desk.ico"
200 ICON "ivan.ico"
```

Целочисленный идентификатор ресурса — это произвольное целое число. При выборе числа нужно помнить, что ресурсы не должны иметь одинаковые идентификаторы.

Обычно программисты присваивают символьные имена числовым значениям. Присваивание в описании ресурсов делается с помощью оператора #define. Программисты на Си могут заметить, что аналогичный оператор имеется в языке Си. Пример выше можно переписать следующим образом:

```
#define ID_ICON1 123
#define ID_ICON2 200
ID_ICON1 ICON "desk.ico"
ID_ICON2 ICON "ivan.ico"
```

В программе на ассемблере можно без проблем использовать целочисленные идентификаторы ресурсов, но для того чтобы можно было использовать символьные имена ID_ICON1 и ID_ICON2, необходимо в секции .data или .const сделать соответствующие присваивания:

```
ID_ICON1 equ 123
ID_ICON2 equ 200
```

Затем можно вызывать функцию LoadIcon, которая возвращает дескриптор иконки в регистр EAX, и заполнять этим значением соответствующие поля структуры WNDCLASSEX:

```
invoke LoadIcon,hInstance,ID_ICON1
mov   wc.hIcon,eax
mov   wc.hIconSm,eax
```

Описание функции LoadIcon можете посмотреть в MSDN.

7.6.2.2. Курсоры

Курсор описывается строкой вида:

```
имя_ID CURSOR имя_файла
```

Где имя_ID — целочисленный идентификатор ресурса

имя_файла — имя файла курсора. Если файл находится не в текущей директории, то необходимо указывать полный путь к файлу.

Пример:

```
1 CURSOR "hand.cur"
2 CURSOR "arrow.cur"
```

Или так:

```
#define ID_CURSOR1 1
#define ID_CURSOR2 2
ID_CURSOR1 CURSOR "hand.cur"
ID_CURSOR2 CURSOR "arrow.cur"
```

Для того чтобы в программе можно было использовать символьные имена ID_CURSOR1 и ID_CURSOR2, необходимо в секции .data или .const сделать соответствующие присваивания:

```
ID_CURSOR1 equ 1
ID_CURSOR2 equ 2
```

Затем можно вызывать функцию LoadCursor, которая возвращает дескриптор иконки в регистр EAX, и заполнять этим значением соответствующие поля структуры WNDCLASSEX:

```
invoke LoadCursor,hInstance,ID_CURSOR1
mov   wc.hCursor,eax
```

Описание функции LoadCursor можете посмотреть в MSDN.

7.6.2.3. Растровые изображения

Растровые изображения описываются подобно иконкам и курсорам строкой вида:

```
имя_ID BITMAP имя_файла
```

Пример:

```
10 BITMAP "ivan.bmp"
13 BITMAP "c:\project\sklyaroff.bmp"
```

Или так:

```
#define ID_ BITMAP1 1
#define ID_ BITMAP2 2
ID_ BITMAP1 BITMAP "ivan.bmp"
ID_ BITMAP2 BITMAP "c:\project\sklyaroff.bmp"
```

Пример использования в программе ресурса с растровым изображением рекомендую посмотреть в рассылке от Iczelion "Урок 25. Простой битмэп" [3].

7.6.2.4. Строки

Строки описываются с помощью следующей структуры, называемой также таблицей строк:

```
STRINGTABLE
{
   имя_ID строка
   . . .
}
```

Ключевое слово STRINGTABLE и фигурные скобки обязательны, даже если описывается всего одна строка. Фигурные скобки могут быть заменены словами BEGIN и END:

```
STRINGTABLE
BEGIN
   имя_ID строка
   . . .
END
```

В отличие от других ресурсов может быть только одна таблица строк.

Пример:

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
{
    IDS_HELLO,   "Hello"
    IDS_GOODBYE, "Goodbye"
}
```

Использовать строки из ресурсов можно с помощью функции LoadString, которая имеет следующее описание:

```
int LoadString(
    HINSTANCE hInstance,
    UINT uID,
    LPTSTR lpBuffer,
    int nBufferMax
);
```

Параметры:

- hInstance — дескриптор экземпляра приложения, получаемый с помощью функции GetModuleHandle;
- uID — номер строки в файле ресурсов;
- lpBuffer — адрес буфера, в который будет помещена строка после выполнения функции;
- nBufferMax — размер буфера.

В секции данных в программе необходимо определить буфер под строку. Например, выделим два буфера для двух строк:

```
BUF1 db 50 dup(0)
BUF2 db 50 dup(0)

; Помещаем первую и вторую строки из ресурсов в соответствующие буфера
invoke LoadString, hInstance, 1, OFFSET BUF1, 50
invoke LoadString, hInstance, 2, OFFSET BUF2, 50

; Теперь строки из буферов можно использовать, например,
; в вызове функции MessageBoxA
invoke MessageBoxA,0,addr BUF1,addr BUF2, MB_OK
```

7.6.2.5. Диалоговые окна

Диалоговые окна описываются с помощью следующей структуры:

```
DialogName DIALOG x, y, width, height
CAPTION "Заголовок окна"
STYLE Стили_диалогового_окна
FONT n, имя_шрифта
{
  Описание элементов диалога
}
```

- DialogName — имя диалогового окна. Диалоговому окну можно присвоить целочисленный идентификатор также как другим ресурсам, однако принято к диалоговым окнам обращаться по именам.
- x, y — координаты верхнего левого угла диалогового окна.
- width, height — ширина и высота диалогового окна.
- CAPTION — название которое будет отображаться в заголовке диалогового окна
- STYLE — описывает стили окна. Здесь можно использовать как стили, применяемые для описания обычных окон, так и стили, применяемые только в диалоговых окнах.

```
ErrorDialog DIALOG 10, 10, 300, 110
STYLE WS_POPUP | WS_BORDER
CAPTION "Error!"
{
 CTEXT "Select One:", 1, 10, 10, 280, 12
 PUSHBUTTON "&Retry", 2, 75, 30, 60, 12
 PUSHBUTTON "&Abort", 3, 75, 50, 60, 12
 PUSHBUTTON "&Ignore", 4, 75, 80, 60, 12
}
```

7.6.2.6. Меню

Перед тем как рассмотреть создание меню вспомним его устройство.

Под заголовком окна приложения располагается меню *верхнего уровня* называемое также *главным меню* (main menu). Меню содержит пункты меню.

Различают два типа пунктов меню:

- пункт-команда — это конечный пункт в иерархическом дереве меню. При выборе пункта-команды Windows посылает процедуре окна сообщение WM_COMMAND, содержащее идентификатор команды, в результате приложение выполняет какое-нибудь действие.

- пункт-подменю — вызывает меню следующего, более низкого уровня (подменю) называемое также *всплывающим меню* (popup-меню).

Главное меню описывается следующей структурой:

```
MenuName MENU  [параметры]
{
Описание всех элементов меню
}
```

MenuName — имя меню. Вместо имени можно использовать целочисленный идентификатор, однако принято к меню, также как и к диалогам, обращаться по имени.

Список элементов меню состоит из выражений MENUITEM и POPUP.

MENUITEM описывает пункт-команду:

```
MENUITEM имя_пункта, MenuID [,параметры]
```

MenuID – идентификатор (произвольное целое число), который будет послан процедуре окна.

Если вместо имя_пункта использовано слово SEPARATOR следующим образом:

```
MENUITEM SEPARATOR
```

то будет отображена горизонтальная разделительная линия вместо пункта меню. Обычно эти горизонтальные линии используются для разделения элементов подменю на группы.

Всплывающее меню описывается следующим образом:

```
POPUP "Имя" [,параметры]
{
Описание всех элементов очередного уровня
}
```

За исключением слова POPUP в заголовке (первая строка) синтаксис и семантика описаний главного меню и подменю совпадает.

Если в имени пункта меню встречается символ "&", то следующий за ним символ на экране будет подчеркнут одинарной чертой. Это означает, что данный элемент меню можно будет вызывать с клавиатуры комбинацией клавиш Alt и подчеркнутого символа.

Меню можно создавать как в обычном, так и в диалоговом окне.

Существует несколько способов подключения ресурса меню к программе.

1. Этот способ связывает одно меню со всеми создаваемыми окнами путем определения его в классе Windows. При регистрации класса полю lpszMenuName структуры WNDCLASS нужно присвоить указатель на строку, содержащую имя меню, например

```
mov wc.lpszMenuName, OFFSET MenuName
```

Когда меню не используется, этому полю присваивается 0.

В диалоговых окнах меню устанавливается иным способом, но этот способ может быть использован и в обычных окнах.

2. Если вы хотите присвоить меню конкретному окну, то можно вначале загрузить меню функцией LoadMenu, прототип этой функции:

```
HMENU LoadMenu(HINSTACE hInstance, LPCTSTR lpMenuName);
```

При успешном выполнении данная функция возвращает дискриптор меню, который затем можно передать функции создания окна CreateWindowEx:

```
.DATA
    MenuName  db "FirstMenu",0
    hMenu HMENU ?
.........................
.........................
.CODE
    .........................
    invoke LoadMenu, hInst, OFFSET MenuName
    mov   hMenu, eax
    invoke CreateWindowEx,NULL,OFFSET ClsName,\
                 OFFSET Caption, WS_OVERLAPPEDWINDOW,\
                CW_USEDEFAULT,CW_USEDEFAULT,\
                CW_USEDEFAULT,CW_USEDEFAULT,\
                NULL,\
                hMenu,\
                hInst,\
                NULL\
    .........................
```

3. Этот способ похож на предыдущий, только после того как дескриптор меню получен функцией LoadMenu, он передается не функции CreateWindowEx, а функции SetMenu, которое устанавливает меню конкретному окну. В функции CreateWindowEx при этом может быть указан NULL в качестве имени меню.

```
BOOL SetMenu(HWND hWnd, HMENU hMenu);
```

В листинге 7.6 показан пример программы с меню с использованием первого способа подключения ресурса меню к программе. Текст файла, содержащий определение меню показан в листинге 7.7.

Компиляция:

```
ml /c /coff /Cp menu.asm
rc menu.rc
link /SUBSYSTEM:WINDOWS /LIBPATH:c:\masm32\lib menu.obj menu.res
```

В процедуре окна WndProc обрабатывается сообщение WM_COMMAND:

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
       .IF uMsg==WM_DESTROY
              invoke PostQuitMessage,NULL
       .ELSEIF uMsg==WM_COMMAND
              mov eax,wParam
              .IF ax==IDM_ABOUT
                     invoke MessageBox,NULL,ADDR About_string,OFFSET
AppName,MB_OK
              .ELSEIF ax==IDM_HELLO
                     invoke MessageBox, NULL,ADDR Hello_string, OFFSET
AppName,MB_OK
              .ELSEIF ax==IDM_GOODBYE
```

```
                invoke MessageBox,NULL,ADDR Goodbye_string, OFFSET
AppName, MB_OK
            .ELSE
                invoke DestroyWindow,hWnd
            .ENDIF
        .ELSE
            invoke DefWindowProc,hWnd,uMsg,wParam,lParam
            ret
        .ENDIF
        xor eax,eax
        ret
WndProc endp
```

При выборе пользователем пункта меню процедуре окна в младшем слове wParam посылается ID пункта меню вместе с сообщением WM_COMMAND. Поэтому, после того как значение wParam сохраняется в EAX (mov eax,wParam), затем сравнивается значение в AX с ID пунктов меню и выводится соответствующее сообщение MessageBox.

Если пользователь выберет пункт "Выход", то будет выполнена API-функция DestroyWindow, которая уничтожит окно.

Листинг 7.6. Программа с меню (menu.asm)

```
        .386
        .model flat,stdcall
        option casemap:none

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

        .data
ClassName     db "SimpleWinClass",0
AppName       db "Our First Window",0
MenuName      db "FirstMenu",0
About_string  db "Пример использования ресурса меню",0
Hello_string  db "Привет, дружище!",0
Goodbye_string        db "Пока, дружище!",0

        .data?
hInstance     HINSTANCE     ?
CommandLine   LPSTR         ?

        .const
IDM_ABOUT     equ 1
IDM_HELLO     equ 2
IDM_GOODBYE   equ 3
IDM_EXIT      equ 4

        .code
```

```
start:
        invoke GetModuleHandle,NULL
        mov    hInstance,eax
        invoke GetCommandLine
        mov    CommandLine,eax
        invoke WinMain,hInstance,NULL,CommandLine,SW_SHOWDEFAULT
        invoke ExitProcess,eax

WinMain proc
        hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
        LOCAL  wc:WNDCLASSEX
        LOCAL  msg:MSG
        LOCAL  hwnd:HWND
        mov    wc.cbSize,SIZEOF WNDCLASSEX
        mov    wc.style,CS_HREDRAW or CS_VREDRAW
        mov    wc.lpfnWndProc,OFFSET WndProc
        mov    wc.cbClsExtra,NULL
        mov    wc.cbWndExtra,NULL
        push   hInst
        pop    wc.hInstance
        mov    wc.hbrBackground,COLOR_WINDOW+1
        mov    wc.lpszMenuName,OFFSET MenuName
        mov    wc.lpszClassName,OFFSET ClassName
        invoke LoadIcon,NULL,IDI_APPLICATION
        mov    wc.hIcon,eax
        mov    wc.hIconSm,eax
        invoke LoadCursor,NULL,IDC_ARROW
        mov    wc.hCursor,eax
        invoke RegisterClassEx,addr wc
        invoke CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
            WS_OVERLAPPEDWINDOW,CW_USEDEFAULT,\
            CW_USEDEFAULT,CW_USEDEFAULT,CW_USEDEFAULT,NULL,NULL,\
            hInst,NULL
        mov    hwnd,eax
        invoke ShowWindow,hwnd,SW_SHOWNORMAL
        invoke UpdateWindow,hwnd
        .WHILE TRUE
                 INVOKE GetMessage,ADDR msg,NULL,0,0
                 .BREAK .IF (!eax)
                 INVOKE DispatchMessage,ADDR msg
        .ENDW
        mov     eax,msg.wParam
        ret
WinMain endp

WndProc proc  hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
        .IF uMsg==WM_DESTROY
               invoke PostQuitMessage,NULL
        .ELSEIF uMsg==WM_COMMAND
               mov    eax,wParam
               .IF ax==IDM_ABOUT
```

```
                invoke MessageBox,NULL,ADDR About_string,OFFSET
AppName,MB_OK
            .ELSEIF ax==IDM_HELLO
                invoke MessageBox,NULL,ADDR Hello_string,OFFSET
AppName,MB_OK
            .ELSEIF ax==IDM_GOODBYE
                invoke MessageBox,NULL,ADDR Goodbye_string,OFFSET
AppName,MB_OK
            .ELSE
                invoke DestroyWindow,hWnd
            .ENDIF
        .ELSE
            invoke DefWindowProc,hWnd,uMsg,wParam,lParam
            ret
        .ENDIF
            xor    eax,eax
            ret
WndProc endp
       end    start
```

Листинг 7.7. Файл с определением ресурса меню (menu.rc)

```
#define IDM_ABOUT    1
#define IDM_HELLO    2
#define IDM_GOODBYE  3
#define IDM_EXIT     4

FirstMenu MENU
{
   POPUP "&File"
        {
         MENUITEM "&Приветствие!",IDM_HELLO
         MENUITEM "&Прощание!",IDM_GOODBYE
         MENUITEM SEPARATOR
         MENUITEM "В&ыход",IDM_EXIT
        }
   MENUITEM "&About",IDM_ABOUT
}
```

7.7. Динамические библиотеки

Динамические библиотеки (DLL — dynamic link libraries) играют большую роль в Windows, можно даже сказать, что на DLL построена сама Windows.

Мы уже пользовались системными библиотеками kernel32.dll, user32.dll, gdi32.dll для вызова системных API функций, однако программист может создавать собственные динамические библиотеки.

DLL — это файлы, которые в отличие от обычных исполняемых файлов, содержат в себе лишь только набор автономных функций для вызова из других приложений или DLL[6].

[6] Динамическая библиотека может содержать также ресурсы.

Зачем нужны динамические библиотеки?

DLL позволяют уменьшить использование памяти и размер исполняемых файлов. Без DLL коды всех используемых программой функций пришлось бы заносить прямо в исполняемый файл. Причем если несколько программ используют одни и те же функции, одинаковые коды функций пришлось бы заносить в каждую программу. DLL позволяет эффективно решить проблему, т. к. DLL загружается в память в единственном экземпляре, а программы, которым нужны функции, подключаются к DLL во время выполнения. При этом в самих программах содержатся только вызовы необходимых функций. Если программа одна использует DLL, то сразу после завершения программы DLL также выгружается из памяти. Но если DLL используют несколько программ, то она останется в памяти, пока ее не выгрузит последняя из использующих ее программ.

Чтобы программа смогла выполнить функцию, находящуюся в библиотеке, необходимо код этой функции сначала загрузить в память вызывающего процесса и передать адрес, по которому находится этот код.

Это можно осуществить двумя способами:

- неявное линкование DLL;
- явная загрузка DLL.

Ниже каждый способ будет рассмотрен подробно, но сначала узнаем, как создать саму динамическую библиотеку.

7.7.1. Простейшая динамическая библиотека

В листинге приведен исходный код простейшей динамической библиотеки. Данная DLL просто выводит сообщение при загрузке библиотеки в память, при ее выгрузке из памяти, а также при вызове функции TestFunction.

В любой динамической библиотеке должна быть определена так называемая "точка входа", т. е. процедура (функция), которая автоматически будет вызываться при загрузке и выгрузке динамической библиотеки. В листинге 7.8 точкой входа является функция DllEntry. Эта функция принимает 3 параметра:

- hInstance — идентификатор DLL, присваиваемый системой;
- reason — причина вызова (значения, которые может принимать этот параметр см. ниже);
- reserved1 — зарезервирован и поэтому равен NULL.

Четыре возможные значения, которые может принимать второй параметр (определены в windows.inc):

```
DLL_PROCESS_DETACH    equ 0
DLL_PROCESS_ATTACH    equ 1
DLL_THREAD_ATTACH     equ 2
DLL_THREAD_DETACH     equ 3
```

- `DLL_PROCESS_ATTACH` — сообщает, что DLL загружается в адресное пространство процесса.
- `DLL_PROCESS_DETACH` — сообщает, что DLL выгружается из адресного пространства процесса.
- `DLL_THREAD_ATTACH` — сообщает, что текущий процесс создает новый поток. Такое сообщение посылается всем динамическим библиотекам, загруженным к этому времени процессом.
- `DLL_THREAD_DETACH` — сообщает, что поток в адресном пространстве процесса уничтожается.

Обратите также внимание, что процедура входа должна возвращать ненулевое значение:

```
mov eax,TRUE
```

Значение TRUE в файле windows.inc объявлено как

```
TRUE equ 1
```

Для компиляции DLL понадобится создать def-файл (листинг 7.9), в котором необходимо просто указать имя библиотеки (после ключевого слова LIBRARY) и

экспортируемые функции (после ключевого слова EXPORTS). Если быть точным DEF-файл необходим для создания статической lib-библиотеки (DLLSkeleton.lib), которая нужна только при неявной загрузке DLL. Для явной загрузки DLL, статическая lib-библиотека не нужна, соответственно в этом случае компилировать DLL можно без DEF-файла.

Следующие две командные строки выполняют компиляцию DLL:

```
ml /c /coff DLLSkeleton.asm
link /SUBSYSTEM:WINDOWS /DLL /DEF:DLLSkeleton.def DLLSkeleton.obj
```

Листинг 7.8. Простейшая DLL-библиотека (DLLSkeleton.asm)

```
        .386
        .model flat,stdcall
        option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

        .data
HelloTitle     db "Пример вызова функции из DLL",0
HelloMsg       db "Это сообщение из DLL",0
LoadMsg        db "DLL загружена",0
UnloadMsg      db "DLL выгружена",0
ThreadCreated  db "Поток создается в этом процессе",0
ThreadDestroyed db "Поток уничтожается в этом процессе",0

        .code
DllEntry PROC hInstance:HINSTANCE,reason:DWORD,reserved1:DWORD
        .if reason==DLL_PROCESS_ATTACH
                invoke MessageBox,NULL,addr LoadMsg,addr HelloTitle,MB_OK
        .elseif reason==DLL_PROCESS_DETACH
                invoke MessageBox,NULL,addr UnloadMsg,addr HelloTitle,MB_OK
        .elseif reason==DLL_THREAD_ATTACH
        invoke MessageBox,NULL,addr ThreadCreated,addr HelloTitle,MB_OK
        .else        ; DLL_THREAD_DETACH
        invoke MessageBox,NULL,addr ThreadDestroyed,addr HelloTitle,MB_OK
        .endif
        mov    eax,TRUE
        ret
DllEntry ENDP

TestFunction PROC
        invoke MessageBox,NULL,addr HelloMsg,addr HelloTitle,MB_OK
        ret
TestFunction ENDP

END DllEntry
```

Листинг 7.9. def-файл (DLLSkeleton.def)

```
LIBRARY DLLSkeleton
EXPORTS TestFunction
```

7.7.2. Неявная загрузка DLL

В листинге 7.10 представлена программа, которая осуществляет неявную загрузку динамической библиотеки, исходный код которой показан в листинге 7.8. Для неявного связывания необходимо с помощью директивы INCLUDELIB подключить статическую библиотеку DLLSkeleton.lib. Эту библиотеку транслятор MASM создает автоматически при компиляции с def-файлом.

Следующие две командные строки выполняют компиляцию программы, демонстрирующую неявную загрузку DLL:

```
ml /c /coff usedll1.asm
link /SUBSYSTEM:WINDOWS usedll1.obj
```

Листинг 7.10. Программа, демонстрирующая неявное линкование DLL (usedll1.asm)

```
        .386
        .model flat,stdcall
        option casemap:none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
includelib DLLSkeleton.lib
includelib \masm32\lib\kernel32.lib

TestFunction PROTO

        .code
start:
        invoke       TestFunction
        invoke       ExitProcess,NULL

       end           start
```

7.7.3. Явная загрузка DLL

В листинге 7.11 представлена программа, которая осуществляет явную загрузку динамической библиотеки, исходный код которой показан в листинге 7.8.

В случае явной загрузки DLL вызывающая программа сама выполняет все манипуляции с DLL. Для того чтобы загрузить библиотеку в память, программа должна вызвать функцию LoadLibrary или LoadLibraryEx.

Функция LoadLibrary принимает единственный аргумент – имя динамической библиотеки, которую требуется загрузить. Описание из MSDN:

```
HMODULE WINAPI LoadLibrary(LPCTSTR lpFileName);
```

Функция LoadLibraryEx является расширенной версией LoadLibrary и принимает три аргумента, ее описание вы можете увидеть в MSDN, а мы обойдемся LoadLibrary.

После загрузки DLL программа должна определить адрес нужной функции в DLL — это делается с помощью функции GetProcAddress. Описание функции:

```
FARPROC WINAPI GetProcAddress(
  HMODULE hModule,
  LPCSTR lpProcName
);
```

Как видно функция GetProcAddress принимает два параметра:

- hModule — дескриптор DLL-библиотеки (полученный с помощью LoadLibrary);
- lpProcName — имя процедуры, которую требуется вызвать из DLL.

В случае успешного выполнения GetProcAddress возвращает адрес процедуры в регистр EAX.

Когда DLL больше не нужна, программа может ее выгрузить с помощью функции FreeLibrary или FreeLibraryAndExitThread. Однако при выходе из программы динамическая библиотека выгружается автоматически системой.

Что касается самой DLL (листинг 7.8), то, разумеется, она никаким изменениям не подвергается.

Следующие две командные строки выполняют компиляцию программы, демонстрирующую явную загрузку DLL:

```
ml /c /coff usedll2.asm
link /SUBSYSTEM:WINDOWS usedll2.obj
```

Листинг 7.11. Программа, демонстрирующая явное линкование DLL (usedll2.asm)

```
        .386
        .model flat,stdcall
        option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

        .data
LibName       db "DLLSkeleton.dll",0
FunctionName  db "TestFunction",0
DllNotFound   db "Не получается загрузить DLL",0
AppName       db "Загружается DLL",0
FunctionNotFound db "Функция TestFunction не найдена",0

        .data?
hLib    dd    ?
TestFunc dd   ?

        .code
start:
        invoke  LoadLibrary,addr LibName
        .if eax==NULL
                invoke MessageBox,NULL,addr DllNotFound,addr
AppName,MB_OK
        .else
              mov hLib,eax
                invoke GetProcAddress,hLib,addr FunctionName
                .if eax==NULL
                        invoke MessageBox,NULL,addr FunctionNotFound,addr
AppName,MB_OK
                .else
                     call eax
                .endif
                invoke FreeLibrary,hLib
        .endif
```

```
        invoke ExitProcess,NULL

      end    start
```

Приложение 1. Основные технические характеристики микропроцессоров фирмы Intel

Таблица П1.1. *Характеристики микропроцессоров Intel*

Модель	Год начала выпуска	Количество транзисторов	Тактовая частота	Разрядность внутренних регистров/ шины данных/ шины адреса	Адресуемая физическая память	Технология производства	Примечания
Первое поколение (186)							
8086	1978	29 тыс.	5-10 МГц	16/16/20	1 Мбайт	3 мкм	
8088	1979	29 тыс.	5-10 МГц	16/8/20	1 Мбайт	3 мкм	
80186	1982	134 тыс.	6 МГц	16/16/20	1 Мбайт	1,5 мкм	Неудавшийся процессор, который не получил широкого распространения
Второе поколение (286)							
80286	1982	134 тыс.	6-12,5 МГц	16/16/24	16 Мбайт	1,5 мкм	Впервые появился защищенный режим и возможность использования виртуальной памяти (1 Гбайт)
Третье поколение (386)							
80386DX	1985	275 тыс.	16-32 МГц	32/32/32	4 Гбайт	1 мкм	Первый 32-разрядный процессор (архитектура IA-32). Появился режим V86 и страничное управление памятью
80386SX	1988	275 тыс.	20-33 МГц	32/16/24	16 Мбайт	1 мкм	
80386SL	1990	275 тыс.	20-25 МГц	32/16/24	16 Мбайт	1 мкм	Спец. модель для мобильных устройств

Таблица П1.1. *(продолжение)*

Четвертое поколение (P4)							
80486DX	1989	1,25 млн.	25-50 МГц	32/32/32	4 Гбайт	1 мкм, 0,8 мкм	Впервые появился встроенный кэш первого уровня (8 Кбайт) и встроенный математичес-кий сопроцессор (FPU) Также начиная с этого процессора стало применяться RISC-ядро
80486SX	1991	0,9 млн.	16-33 МГц	32/16/24	16 Мбайт	0,8 мкм	
80486SL	1992	1,25 млн.	25-33 МГц	32/32/32	4 Гбайт		
80486DX 2	1992	1,25 млн.	50-66 МГц	32/32/32	4 Гбайт		
80486SX2	1992	0,9 млн.	50 МГц	32/16/24	16 Мбайт		
80486DX 4	1994	1,6 млн.	75-100 МГц	32/32/32	4 Гбайт		
Пятое поколение (P5)							
Pentium	1993	3,1-3,3 млн.	60-200 МГц	32/64/32	64 Гбайт	0,8 мкм, 0,6 мкм, 0,35 мкм	Первый процессор с суперскаляр-ной (двухконвей-ерной) архитектурой
Pentium MMX	1997	4,5 млн.	166-233 МГц	32/64/32	64 Гбайт	0,35 мкм	Изобретено расширение MMX (Multi Media eXtention), содержащее 57 инструкций для вычислений с плавающей точкой, существенно увеличиваю-щее производите-льность компьютера в мультимедиа-приложениях

Таблица П1.1. *(окончание)*

Шестое поколение (P6)							
Pentium Pro	1995	5,5 млн.	150-200 МГц	32/64/32	64 Гбайт	0,5 мкм, 0,35 мкм	Впервые применена кэш-память второго уровня, работающая на частоте ядра процессора (256 Кбайт).
Pentium II	1997	7,5 млн.	233-450 МГц	32/64/64	64 Гбайт	0,35 мкм, 0,25 мкм	
Pentium III	1999	9,5-28,1 млн.	450-1200 МГц	32/64/64	64 Гбайт	0,25 мкм	Появился блок SSE (Streaming SIMD Extensions)
Celeron	1998	7,5-19 млн.	266 МГц -2,80 ГГц	32/64/64	4 Гбайт	0,25 мкм	
Xeon	2000	9,5 млн.	1,40 ГГц - 3,66 ГГц	32/64/64	64 Гбайт	0,25 мкм, 0,18 мкм, 0,13 мкм	Процессор для многопроцессорных производительных рабочих станций и серверов
Седьмое поколение (P7)							
Pentium 4	2000	42-55 млн.	1,4 ГГц -3,80 ГГц	32/64/64	64 Гбайт	0,18 мкм, 0,13 мкм	Впервые введен гиперконвейер (более 20-и ступеней) Появились потоковые SIMD-расширения 2 (SSE2)

Приложение 2. Таблицы кодов символов

ASCII (American Standard Code for Information Interchange — американский стандартный код для обмена информацией) — используется для представления символов в виде чисел. ASCII является 8-битной кодировкой, поэтому может содержать максимум 256 символов. Этого не достаточно для хранения всех существующих символов и букв национальных алфавитов, поэтому кодовая таблица ASCII разделяется на две половины:

- основная (постоянная) — содержит коды в диапазоне от 0 до 127 и используется для представления цифр, букв латинского алфавита, знаков препинания и других символов.
- дополнительная (изменяемая) — содержит коды в диапазоне от 128 до 255 и используется для представления букв национальных алфавитов и символов псевдографики.

В таблице П2.1 показаны основная таблица ASCII, а в таблицах П2.2-2.6 наиболее распространенные в России кодировки, использующиеся в качестве дополнительной таблицы ASCII.

Кодировкой по умолчанию для второй половины ASCII является CP-437, которая использовалась на самых первых компьютерах и сейчас используется BIOS.

В системе MS-DOS, а также в консольных окнах Windows в качестве дополнительной таблицы ASCII обычно используется кодировка CP-866. В графических приложениях Windows обычно используется кодировка CP-1251. В системах UNIX обычно используются кодировки KOI8-R и ISO 8859-5.

Первые 32 кода основной таблицы могут также использоваться как управляющие коды (таблица П2.7).

Таблица П2.1. *Основные коды ASCII (1-128)*

0		16	►	32		48	0	64	@	80	P	96	`	112	p
1	☺	17	◄	33	!	49	1	65	A	81	Q	97	a	113	q
2	☻	18	↕	34	"	50	2	66	B	82	R	98	b	114	r
3	♥	19	‼	35	#	51	3	67	C	83	S	99	c	115	s
4	♦	20	¶	36	$	52	4	68	D	84	T	100	d	116	t
5	♣	21	§	37	%	53	5	69	E	85	U	101	e	117	u
6	♠	22	▬	38	&	54	6	70	F	86	V	102	f	118	v
7	•	23	↨	39	'	55	7	71	G	87	W	103	g	119	w
8	◘	24	↑	40	(	56	8	72	H	88	X	104	h	120	x
9	○	25	↓	41	)	57	9	73	I	89	Y	105	i	121	y
10	◙	26	→	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	27	←	43	+	59	;	75	K	91	[	107	k	123	{
12	♀	28	∟	44	,	60	<	76	L	92	\	108	l	124	|
13	♪	29	↔	45	-	61	=	77	M	93	]	109	m	125	}
14	♫	30	▲	46	.	62	>	78	N	94	^	110	n	126	~
15	☼	31	▼	47	/	63	?	79	O	95	_	111	o	127	⌂

Таблица П2.2. *Дополнительные коды ASCII (кодировка ср437)*

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▓	193	┴	209	╤	225	β	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	·
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	∅	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238		254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Таблица П2.3. *Дополнительные коды ASCII 128-255 (кодировка ср866)*

128	А	144	Р	160	а	176	▒	192	└	208	╨	224	р	240	Ё
129	Б	145	С	161	б	177	▒	193	┴	209	╤	225	с	241	ё
130	В	146	Т	162	в	178	▓	194	┬	210	╥	226	т	242	Є
131	Г	147	У	163	г	179	│	195	├	211	╙	227	у	243	є
132	Д	148	Ф	164	д	180	┤	196	─	212	╘	228	ф	244	Ї
133	Е	149	Х	165	е	181	╡	197	┼	213	╒	229	х	245	ї
134	Ж	150	Ц	166	ж	182	╢	198	╞	214	╓	230	ц	246	Ў
135	З	151	Ч	167	з	183	╖	199	╟	215	╫	231	ч	247	ў
136	И	152	Ш	168	и	184	╕	200	╚	216	╪	232	ш	248	°
137	Й	153	Щ	169	й	185	╣	201	╔	217	┘	233	щ	249	·
138	К	154	Ъ	170	к	186	║	202	╩	218	┌	234	ъ	250	·
139	Л	155	Ы	171	л	187	╗	203	╦	219	█	235	ы	251	√
140	М	156	Ь	172	м	188	╝	204	╠	220	▄	236	ь	252	№
141	Н	157	Э	173	н	189	╜	205	═	221	▌	237	э	253	¤
142	О	158	Ю	174	о	190	╛	206	╬	222	▐	238	ю	254	■
143	П	159	Я	175	п	191	┐	207	╧	223	▀	239	я	255	

Таблица П2.4. *Дополнительные коды ASCII (кодировка cp1251)*

128	Ђ	144	ђ	160		176	°	192	А	208	Р	224	а	240	р
129	Ѓ	145	‘	161	Ў	177	±	193	Б	209	С	225	б	241	с
130	‚	146	’	162	ў	178	І	194	В	210	Т	226	в	242	т
131	ѓ	147	“	163	Ј	179	і	195	Г	211	У	227	г	243	у
132	„	148	”	164	¤	180	ґ	196	Д	212	Ф	228	д	244	ф
133	…	149	•	165	Ґ	181	µ	197	Е	213	Х	229	е	245	х
134	†	150	–	166	¦	182	¶	198	Ж	214	Ц	230	ж	246	ц
135	‡	151	—	167	§	183	·	199	З	215	Ч	231	з	247	ч
136	€	152		168	Ё	184	ё	200	И	216	Ш	232	и	248	ш
137	‰	153	™	169	©	185	№	201	Й	217	Щ	233	й	249	щ
138	Љ	154	љ	170	Є	186	є	202	К	218	Ъ	234	к	250	ъ
139	‹	155	›	171	«	187	»	203	Л	219	Ы	235	л	251	ы
140	Њ	156	њ	172	¬	188	j	204	М	220	Ь	236	м	252	ь
141	Ќ	157	ќ	173	-	189	S	205	Н	221	Э	237	н	253	э
142	Ћ	158	ћ	174	®	190	s	206	О	222	Ю	238	о	254	ю
143	Џ	159	џ	175	Ї	191	ї	207	П	223	Я	239	п	255	я

Таблица П2.5. *Дополнительные коды ASCII (кодировка KOI8-R)*

128	─	144	░	160	═	176	╠	192	ю	208	п	224	Ю	240	П
129	│	145	▒	161	‖	177	╣	193	а	209	я	225	А	241	Я
130	┌	146	▓	162	╒	178	╡	194	б	210	р	226	Б	242	Р
131	┐	147	⌠	163	ё	179	Ё	195	ц	211	с	227	Ц	243	С
132	└	148	■	164	╓	180	╢	196	д	212	т	228	Д	244	Т
133	┘	149	·	165	╔	181	╖	197	е	213	у	229	Е	245	У
134	├	150	√	166	╕	182	╤	198	ф	214	ж	230	Ф	246	Ж
135	┤	151	≈	167	╗	183	╥	199	г	215	в	231	Г	247	В
136	┬	152	≤	168	╘	184	╦	200	х	216	ь	232	Х	248	Ь
137	┴	153	≥	169	╙	185	╧	201	и	217	ы	233	И	249	Ы
138	┼	154		170	╚	186	╨	202	й	218	з	234	Й	250	З
139	▀	155	⌡	171	╛	187	╩	203	к	219	ш	235	К	251	Ш
140	▄	156	°	172	╜	188	╪	204	л	220	э	236	Л	252	Э
141	█	157	²	173	╝	189	╫	205	м	221	щ	237	М	253	Щ
142	▌	158	·	174	╛	190	╬	206	н	222	ч	238	Н	254	Ч
143	▐	159	÷	175	╞	191	©	207	о	223	ъ	239	О	255	Ъ

Таблица П2.6. *Дополнительные коды ASCII (кодировка ISO 8859-5)*

128	144	160		176	А	192	Р	208	а	224	р	240	№
129	145	161	Ё	177	Б	193	С	209	б	225	с	241	ё
130	146	162	Ђ	178	В	194	Т	210	в	226	т	242	ђ
131	147	163	Ѓ	179	Г	195	У	211	г	227	у	243	ѓ
132	148	164	Є	180	Д	196	Ф	212	д	228	ф	244	є
133	149	165	S	181	Е	197	Х	213	е	229	х	245	s
134	150	166	I	182	Ж	198	Ц	214	ж	230	ц	246	i
135	151	167	Ї	183	З	199	Ч	215	з	231	ч	247	ї
136	152	168	J	184	И	200	Ш	216	и	232	ш	248	j
137	153	169	Љ	185	Й	201	Щ	217	й	233	щ	249	љ
138	154	170	Њ	186	К	202	Ъ	218	к	234	ъ	250	њ
139	155	171	Ћ	187	Л	203	Ы	219	л	235	ы	251	ћ
140	156	172	Ќ	188	М	204	Ь	220	м	236	ь	252	ќ
141	157	173	-	189	Н	205	Э	221	н	237	э	253	§
142	158	174	Ў	190	О	206	Ю	222	о	238	ю	254	ў
143	159	175	Џ	191	П	207	Я	223	п	239	я	255	џ

Таблица П2.7. *Управляющие символы ASCII*

Знак	**Имя**	**Код**		**Ctrl-код**	**Назначение**
		(дес.)	**(шестн.)**		
	NUL	0	00h	^@	Пусто
☺	SOH	1	01h	^A	Начало заголовка
☻	STX	2	02h	^B	Начало текста
♥	ETX	3	03h	^C	Конец текста
♦	EOT	4	04h	^D	Конец передачи
♣	ENQ	5	05h	^E	Запрос
♠	ACK	6	06h	^F	Подтверждение
•	BEL	7	07h	^G	Звонок
◘	BS	8	08h	^H	Шаг назад
○	HT	9	09h	^I	Горизонтальная табуляция
◙	LF	10	0Ah	^J	Перевод строки
♂	VT	11	0Bh	^K	Вертикальная табуляция
♀	FF	12	0Ch	^L	Подача формы
♪	CR	13	0Dh	^M	Возврат каретки
♫	SO	14	0Eh	^N	Сдвиг с исключением
☼	SI	15	0Fh	^O	Сдвиг с включением
►	DLE	16	10h	^P	Оставить канал данных

Таблица П2.7. *(окончание)*

◄	DC1/XON	17	11h	^Q	Управление устройством 1
↕	DC2	18	12h	^R	Управление устройством 2
‼	DC3/XOFF	19	13h	^S	Управление устройством 3
¶	DC4	20	14h	^T	Управление устройством 4
§	NAK	21	15h	^U	Отрицательное подтверждение
▬	SYN	22	16h	^V	Синхронизация
↨	ETB	23	17h	^W	Конец блока передачи
↑	CAN	24	18h	^X	Отмена
↓	EM	25	19h	^Y	Конец носителя
→	SUB	26	1Ah	^Z	Замена
←	ESC	27	1Bh	^[	Escape
∟	FS	28	1Ch	^\	Разделитель файлов
↔	GS	29	1Dh	^]	Разделитель групп
▲	RS	30	1Eh	^^	Разделитель записей
▼	US	31	1Fh	^_	Разделитель полей
	SP	32	20		Пробел
!	DEL	33	7F	^?	Удаление

Таблица П2.8. *Расширенные ASCII-коды*[7]

Клавиша	**Код**	**Клавиша**	**Код**	**Клавиша**	**Код**	**Клавиша**	**Код**	**Клавиша**	**Код**
F1	3Bh	Alt-R	13h	Shift-F11	87h	Alt-Tab	A5h	Alt-I	17h
F2	3Ch	Alt-S	1Fh	Shift-F12	88h	Ctrl-Tab	94h	Alt-J	24h
F3	3Dh	Alt-T	14h	Alt-0	81h	Alt-Del	A3h	Alt-K	25h
F4	3Eh	Alt-U	16h	Alt-1	82h	Alt-End	9Fh	Alt-L	26h
F5	3Fh	Alt-V	2Fh	Alt-2	83h	Alt-Home	97h	Ctrl-Right	74h
F6	40h	Alt-W	11h	Alt-3	84h	Alt-Ins	A2h	Ctrl-End	75h
F7	41h	Alt-X	2Dh	Alt-4	85h	Alt-PgUp	99h	Ctrl-Home	77h
F8	42h	Alt-Y	15h	Alt-5	86h	Alt-PgDn	A1h	Ctrl-PgDn	76h
F9	43h	Alt-Z	2Ch	Alt-6	87h	Alt-Enter	1Ch	Ctrl-PgUp	84h
F10	44h	Alt-\	2Bh	Alt-7	88h	Ctrl-F1	5Eh	Alt-Up	98h
F11	85h	Alt-,	33h	Alt-8	89h	Ctrl-F2	5Fh	Alt-Down	A0h
F12	86h	Alt-.	34h	Alt-9	8Ah	Ctrl-F3	60h	Alt-Left	9Bh
Alt-F1	68h	Alt-/	35h	AltC	8Bh	Ctrl-F4	61h	Alt-Right	9Dh
Alt-F2	69h	Alt-BS	0Eh	Alt-=	8Ch	Ctrl-F5	62h	Alt-K/	A4h
Alt-F3	6Ah	Alt-[	1Ah	NUL	03h	Ctrl-F6	63h	Ctrl-K*	37h
Alt-F4	6Bh	Alt-]	1Bh	Shift-Tab	0Fh	Ctrl-F7	64h	Alt-K-	4Ah
Alt-F5	6Ah	Alt-;	27h	Ins	52h	Ctrl-F8	65h	Alt-K+	4Eh
Alt-F6	6Dh	Alt-'	28h	Del	53h	Ctrl-F9	66h	Alt-KEnter	A6h
Alt-F7	6Eh	Alt-`	29h	SysRq	72h	Ctrl-F10	67h	Ctrl-K/	95h
Alt-F8	6Fh	Shift-F1	54h	Down	50h	Ctrl-F11	89h	Ctrl-K*	96h

[7] Префикс "K" соответствует клавишам цифровой клавиатуры

Таблица П2.8. *(окончание)*

Alt-F9	70h	Shift-F2	55h	Left	4Bh	Ctrl-F12	8Ah	Ctrl-K-	8Eh
Alt-F10	71h	Shift-F3	56h	Right	4Dh	Alt-A	1Eh	Ctrl-K+	90h
Alt-F11	8Bh	Shift-F4	57h	Up	48h	Alt-B	30h	Ctrl-K8	8Dh
Alt-F12	8Ch	Shift-F5	58h	Enter	4Fh	Alt-C	2Eh	Ctrl-K5	8Fh
Alt-M	32h	Shift-F6	59h	Home	47h	Alt-D	20h	Ctrl-K2	91h
Alt-N	31h	Shift-F7	5Ah	PgDn	51h	Alt-E	12h	Ctrl-K0	92h
Alt-O	18h	Shift-F8	5Bh	PgUp	49h	Alt-F	21h	Ctrl-K	93h
Alt-P	19h	Shift-F9	5Ch	Ctrl-Left	73h	Alt-G	22h		
Alt-Q	10h	Shift-F10	5Dh	Alt-Esc	01h	Alt-H	23h		

Таблица П2.9. *Скан-коды клавиш*

Клавиша	**Make (HEX)**	**Break (HEX)**	**Клавиша**	**Make (HEX)**	**Break (HEX)**	**Клавиша**	**Make (HEX)**	**Break (HEX)**
Esc	01h	81h	Y	15h	95h	K*	37h	B7h
1 !	02h	82h	Z	2Ch	ACh	K-	4Ah	CAh
2 @	03h	83h	; :	27h	A7h	K+	4Eh	CEh
3 #	04h	84h	' "	28h	A8h	K/	35h	B5h
4 $	05h	85h	` ~	29h	A9h	K0	52h	D2h
5 %	06h	86h	\ |	2Bh	ABh	K1	4Fh	CFh
6 ^	07h	87h	, <	33h	B3h	K2	50h	D0h
7 &	08h	88h	. >	34h	B4h	K3	51h	D1h
8 *	09h	89h	/ ?	35h	B5h	K4	4Bh	CBh
9 (	0Ah	8Ah	[{	1Ah	9Ah	K5	4Ch	CCh
0)	0Bh	8Bh	] }	1Bh	9Bh	K6	4Dh	CDh
- _	0Ch	8Ch	Enter	1Ch	9Ch	K7	47h	C7h
= +	0Dh	8Dh	Ctrl	1Dh	9Dh	K8	48h	C8h
BS	0Eh	8E	RShift	36h	B6h	K9	49h	C9h
Tab	0Fh	8Fh	LShift	2Ah	AAh	F1	3Bh	BBh
A	1Eh	9Eh	Num	45h	C5h	F2	3Ch	BCh
B	30h	B0h	Scroll	46h	C6h	F3	3Dh	BDh
C	2Eh	AEh	Home	47h	AAh	F4	3Eh	BEh
D	20h	A0h	End	4Fh	C7h	F5	3Fh	BFh
E	12h	92h	Ins	52h	D2h	F6	40h	C0h
F	21h	A1h	Del	53h	D3h	F7	41h	C1h
G	22h	A2h	-	48h	C8h	F8	42h	C2h
H	23h	A3h	PgUp	49h	C9h	F9	43h	C3h
I	17h	97h	PgDn	51h	D1h	F10	44h	C4h
J	24h	A4h	Alt	38h	B8h	F11	57h	D7h
K	25h	A5h	SP	39h	B9h	F12	58h	D8h
L	26h	A6h	Caps	3Ah	BAh	F13/LWin	5Bh	DBh
M	32h	B2h	SysRq	54h	D4h	F14/RWin	5Ch	DCh

Таблица П2.9. *(окончание)*

N	31h	B1h	Macro	56h	D6h	F15/Menu	5Dh	DDh
O	18h	98h		4Bh	CBh	F16	63h	E3h
P	19h	99h	®	4Dh	CDh	F17	64h	E4h
Q	10h	90h	Ï	50h	D0h	F18	65h	E5h
R	13h	93h	PA1	5Ah	DAh	F19	66h	E6h
S	1Fh	9Fh	EraseEOF	6Dh	EDh	F20	67h	E7h
T	14h	94h	Copy/Play	6Fh	EFh	F21	68h	E8h
U	16h	96h	CrSel	72h	F2h	F22	69h	E9h
V	2Fh	AFh	Delta	73h	F3h	F23	6Ah	EAh
W	11h	91h	ExSel	74h	F4h	F24	6Bh	EBh
X	2Dh	ADh	Clear	76h	F6h			

Таблица П2.10. *Служебные скан-коды*

Код	**Функция**
00h	Буфер клавиатуры переполнен
AAh	Самотестирование закончено
E0h	Префикс для серых клавиш
E1h	Префикс для клавиш без кода отпускания
F0h	Префикс отпускания клавиши
EEh	Эхо
FAh	ACK
FCh	Ошибка самотестирования
FDh	Ошибка самотестирования
FEh	RESEND
FFh	Ошибка клавиатуры

Приложение 3. Сравнение двух синтаксисов ассемблера

Синтаксис AT&T характерен для ассемблеров Unix-систем AS и GAS. Ассемблеры MASM, MASM32, TASM, NASM, FASM и большинство других используют Intel-синтаксис.

Современные версии GAS поддерживают директиву `.intel_syntax`, которая позволяет использовать Intel-синтаксис в GAS.

Таблица П3.1. *Сравнение двух синтаксисов ассемблера, с примерами кода*

Intel Syntax	AT&T Syntax
Регистры записываются без каких-либо префиксов: eax, ebx, ecx, ...	Перед регистрами всегда ставится знак процента: %eax, %abx, %ecx, …

Перед непосредственными операндами не указывается никаких символов:
`push 1`
`sub esp,50h`
	Перед непосредственными операндами указывается символ "$":
	`push $1`
	`sub $0x50,%esp`

Для обозначения шестнадцатеричного числа в инструкциях используется суффикс h: `int 80h`	Для обозначения шестнадцатеричного числа в инструкциях используется префикс "0x": `int $0x80`

В командах с несколькими операндами, первым указывается приемник, а последним — источник:
`mov eax,1`
`imul eax,edx,13`
	В командах с несколькими операндами, первым указывается источник, а последним — приемник, то есть в точности, наоборот, по сравнению с Intel-синтаксисом:
	`movb $1,%eax`
	`imul $13,%edx,%eax`

Для отражения размера операндов используются директивы: byte ptr – байт, word ptr – слово, dword ptr – двойное слово. Примеры:
`mov byte ptr var1,1`
`mov word ptr var2,100`
`push dword ptr var3`
	К названиям команд добавляются суффиксы, отражающие размер операндов: b – байт, w – слово, l – двойное слово. Примеры:
	`movb $1,var1`
	`movw $100,var2`
	`pushl var3`

Для объявления 32-, 16- и 8-разрядных чисел используются директивы dd, dw и db соответственно:
`var1 db 5Ah`
`var2 dw 0`
`var3 dd 30`
`str db "строка символов", 0`
	Для объявления 32-, 16- и 8-разрядных чисел используются директивы .int (или .long), .word и .byte соответственно, при этом переменные объявляются так же как метки (с помощью двоеточия):
	`var1: .byte 0x5A`
	`var2: .word 0`
	`var3: .int 30`
	Также используются другие директивы: .ascii или .string – для объявления строки байтов, .asciz – для объявления строки байтов с автоматически добавляемым нулем в конце, .float – 32-битные числа с плавающей запятой, .double – 64-битные числа с плавающей запятой и др.
	`str: .asciz "строка символов"`

Для разыменования значения по указанному адресу в памяти, используются квадратные скобки: [var1]	Для разыменования значения по указанному адресу в памяти, используются круглые скобки: (var1)

Таблица П3.1. *(окончание)*

Для адресации переменной со смещением лежащем в регистре, прибавляется или вычитается необходимое количество байт внутри квадратных скобок:
`mov ebx,[ebp + 8]`
`mov edx,[ebp - 4]`
	Для адресации переменной со смещением лежащем в регистре, необходимое количество прибавляемых или вычитаемых байт указывается до круглых скобок:
	`movl 8(%ebp),%ebx`
	`movl -4(%ebp),%edx`

Косвенная адресация имеет форму:
`СЕКЦИЯ:[БАЗА + ИНДЕКС*МАСШТАБ + СМЕЩЕНИЕ]`
Примеры:
`mov eax,base_addr[ebx+edi*3]`
`mov eax,[var1+esi*4]`
	Косвенная адресация имеет форму:
	`%СЕКЦИЯ:СМЕЩЕНИЕ(БАЗА, ИНДЕКС, МАСШТАБ)`
	Примеры:
	`movl base_addr(%ebx,%edi,3),%eax`
	`movl var1(,%esi,4),%eax`

Длинные переходы и вызовы записываются как: `call/jmp far SECTION:OFFSET`	Длинные переходы и вызовы записываются как: `lcall/ljmp $SECTION,$OFFSET`

Комментарии должны начинаться с символа ";" и записываться в одну строку:
`; Это комментарий Intel-синтаксиса,`
`; каждая строка должна начинаться`
`; с символа ";".`
	Поддерживаются комментарии в стиле C (/* */), в стиле C++ (//) и в стиле shell (#):
	`# Это комментарий AT&T-синтаксиса.`
	`/* Это многострочный комментарий`
	`AT&T-синтаксиса. */`
	`//Это тоже комментарий в стиле AT&T`

Список литературы

1. Гук М., Юров В. Процессоры Pentium, Athlon и Duron. — СПб.: Питер, 2001.
2. Зубков С. Assembler для DOS, Windows и UNIX. — М., 2000.
3. Туториалы Iczelion'a. **http://wasm.ru**.
4. Румянцев П. В. Азбука программирования в Win32 API. — 4-е изд. — М.: Горячая линия – Телеком, 2004.
5. Румянцев П. В. Работа с файлами в Win32 API. — 2-е изд. — М.: Горячая линия – Телеком, 2002.