Информация об авторе:

Автор: Поляков Андрей Валерьевич
Web: http://av-assembler.ru
e-mail: avprog@narod.ru

Страница книги: http://av-assembler.ru/asm/afd/assembler-for-dummy.htm

1. РАЗРЕШЕНИЯ

Разрешается использование книги в ознакомительных и образовательных целях, а также бесплатное распространение книги, если это не противоречит правилам раздела «2. ОГРАНИЧЕНИЯ».

2. ОГРАНИЧЕНИЯ

Запрещается использование книги в коммерческих целях (продажа, включение в состав платных продуктов и т.п.). Запрещается размещение книги на любых Интернет-ресурсах. Запрещается вносить изменения в текст книги.

Данная книга является бесплатной. Однако если вы захотите выразить благодарность автору за его труды в виде денежного вознаграждения, то можете перечислить деньги на любой из следующих счетов:

WebMoney:
R195975572634
Z167111238420
B180729885812

Яндекс.Деньги:
4100188102149

Я не гордый, с благодарностью приму любую сумму ☺

Если же на вашем счёте унылые нули или у вас вообще нет электронного кошелька, то создайте электронный кошелёк и **заработайте свои первые деньги в Интернете**. Как это сделать, описано на сайте:

http://how-get-wm.narod.ru

А потратить эти деньги вы можете здесь: **Электронный магазин**

Ссылки

http://www.tz-5133.narod.ru
Всё для студента: Методички, книги, статьи, программы, рефераты, контрольные, курсовые и прочая полезная информация.

http://www.avprog.narod.ru
Автоматизация, программирование, телефония, электроника и другая полезная информация.

http://www.av-physics.narod.ru
Интерактивный учебник по физике.

http://av-photography.narod.ru
Фотографии, которые можно использовать как обои для рабочего стола. Также есть описание бесплатного графического редактора GIMP.

"Чайникам" о компьютерах
Книга о компьютерах для начинающих.

Интернет для начинающих
Книга об Интернете для начинающих.

Как стать программистом?
Книга о том, как писать программы и что такое программы вообще.

Заработок в Интернете как точная наука
Книга о том, можно ли заработать в Интернете, и если можно, то как и сколько.

Поляков А.В.

Ассемблер для чайников

2011 г.

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Ассемблер – это магическое слово взывает благоговейный трепет у начинающих программистов. Общаясь между собой, они обязательно говорят о том, что где-то у кого-то есть знакомый «чувак», который может читать исходные коды на языке ассемблера как книжный текст. При этом, как правило, язык ассемблера воспринимается как нечто недоступное простым смертным.

Отчасти это действительно так. Можно выучить несколько простых команд и даже написать какую-нибудь программку, но настоящим гуру (в любом деле) можно стать только в том случае, когда человек очень хорошо знает теоретические основы и понимает, что и зачем он делает.

Есть другая крайность – бывалые программисты на языках высокого уровня убеждены, что **язык ассемблера** – это пережиток прошлого. Да, средства разработки за последние 20 лет шагнули далеко вперёд. Теперь можно написать простенькую программу вообще не зная ни одного языка программирования. Однако не стоит забывать о таких вещах, как, например, микроконтроллеры. Да и в компьютерном программировании некоторые задачи проще и быстрее решить с помощью языка ассемблера.

Данная книга предназначена для тех, кто уже имеет навыки программирования на языке высокого уровня, но хотел бы перейти «ближе к железу» и разобраться с тем, как выполняются команды процессора, как происходит распределение памяти, как управляются разные «железяки» типа дисководов и т.п.

Книга разбита на несколько разделов. Первый раздел – быстрый старт. Здесь очень кратко описаны основные принципы программирования на языке Ассемблера, сами ассемблеры (компиляторы) и методы работы с ассемблерами. Если вы уверенно себя чувствуете в программировании на высоком уровне, но хотели бы освоить азы низкоуровневого программирования, то, быть может, вам будет достаточно прочитать только этот раздел.

Второй раздел описывает такие вещи, как системы исчисления, представления данных в памяти компьютера и т.п., то есть вещи, которые непосредственно к программированию не относятся, но без которых профессиональное программирование невозможно. Также во втором разделе более подробно рассматриваются общие принципы программирования на языке Ассемблера.

Остальные разделы описывают некоторые конкретные примеры программирования на языке Ассемблера, содержат справочные материалы и т.п.

Основы программирования вообще в этой книге не описаны, поэтому для начинающих настоятельно рекомендую ознакомиться с книгой Как стать программистом, где разъяснены «на пальцах» общие принципы программирования и подробно рассмотрены примеры создания простых программ от программ для компьютеров до программ для станков с ЧПУ.

ВВЕДЕНИЕ

Для начала разберёмся с терминологией.

Машинный код – система команд конкретной вычислительной машины (процессора), которая интерпретируется непосредственно процессором. Команда, как правило, представляет собой целое число, которое записывается в регистр процессора. Процессор читает это число и выполняет операцию, которая соответствует этой команде. Популярно это описано в книге Как стать программистом.

Язык программирования низкого уровня (низкоуровневый язык программирования) – это язык программирования, максимально приближённый к программированию в машинных кодах. В отличие от машинных кодов, в языке низкого уровня каждой команде соответствует не число, а сокращённое название команды (мнемоника). Например, команда ADD – это сокращение от слова ADDITION (сложение). Поэтому использование языка низкого уровня существенно упрощает написание и чтение программ (по сравнению с программированием в машинных кодах). Язык низкого уровня привязан к конкретному процессору. Например, если вы написали программу на языке низкого уровня для процессора PIC, то можете быть уверены, что она не будет работать с процессором AVR.

Язык программирования высокого уровня – это язык программирования, максимально приближённый к человеческому языку (обычно к английскому, но есть языки программирования на национальных языках, например, язык 1С основан на русском языке). Язык высокого уровня практически не привязан ни к конкретному процессору, ни к операционной системе (если не используются специфические директивы).

Язык ассемблера – это низкоуровневый язык программирования, на котором вы пишите свои программы. Для каждого процессора существует свой язык ассемблера.

Ассемблер – это специальная программа, которая преобразует (ассемблирует, то есть собирает) исходные тексты вашей программы, написанной на языке ассемблера, в исполняемый файл (файл с расширением EXE или COM). Если быть точным, то для создания исполняемого файла требуются дополнительные программы, а не только ассемблер. Но об этом позже…

В большинстве случаев говорят «ассемблер», а подразумевают «язык ассемблера». Теперь вы знаете, что это разные вещи и так говорить не совсем правильно. Хотя все программисты вас поймут.

ВАЖНО!
В отличие от языков высокого уровня, таких, как Паскаль, Бейсик и т.п., для КАЖДОГО АССЕМБЛЕРА существует СВОЙ ЯЗЫК АССЕМБЛЕРА. Это правило в корне отличает язык ассемблера от языков высокого уровня. Исходные тексты программы (или просто «исходники»), написанной на языке высокого уровня, вы в большинстве случаев можете откомпилировать разными компиляторами для разных процессоров и разных операционных систем. С ассемблерными исходниками это сделать будет намного сложнее. Конечно, эта разница почти не ощутима для разных ассемблеров, которые предназначены для одинаковых процессоров. Но в том то и дело, что для КАЖДОГО ПРОЦЕССОРА существует СВОЙ АССЕМБЛЕР и СВОЙ ЯЗЫК АССЕМБЛЕРА. В этом смысле программировать на языках высокого уровня гораздо проще. Однако за все удовольствия надо платить. В случае с языками высокого уровня мы можем столкнуться с такими вещами как больший размер исполняемого файла, худшее быстродействие и т.п.

В этой книге мы будем говорить только о программировании для компьютеров с процессорами Intel (или совместимыми). Для того чтобы на практике проверить приведённые в книге примеры, вам потребуются следующие программы (или хотя бы некоторые из них):

1. **Emu8086**. Хорошая программа, особенно для новичков. Включает в себя редактор исходного кода и некоторые другие полезные вещи. Работает в Windows, хотя программы пишутся под DOS. К сожалению, программа стоит денег (но оно того стоит))). Подробности см. на сайте http://www.emu8086.com.
2. **TASM** – Турбо Ассемблер от фирмы Borland. Можно создавать программы как для DOS так и для Windows. Тоже стоит денег и в данный момент уже не поддерживается (да и фирмы Borland уже не существует). А вообще вещь хорошая.
3. **MASM** – Ассемблер от компании Microsoft (расшифровывается как МАКРО ассемблер, а не Microsoft Assembler, как думают многие непосвящённые). Пожалуй, самый популярный ассемблер для процессоров Intel. Поддерживается до сих пор. Условно бесплатная программа. То есть, если вы будете покупать её отдельно, то она будет стоить денег. Но она доступна бесплатно подписчикам MSDN и входит в пакет программ Visual Studio от Microsoft.
4. **WASM** – ассемблер от компании Watcom. Как и все другие, обладает преимуществами и недостатками.
5. **Debug** - обладает скромными возможностями, но имеет большой плюс - входит в стандартный набор Windows. Поищите ее в папке WINDOWS\COMMAND или WINDOWS\SYSTEM32. Если не найдете, тогда в других папках каталога WINDOWS.
6. Желательно также иметь какой-нибудь **шестнадцатеричный редактор**. Не помешает и досовский файловый менеджер, например Волков Коммандер (VC) или Нортон Коммандер (NC). С их помощью можно также посмотреть шестнадцатеричные коды файла, но редактировать нельзя. Бесплатных шестнадцатеричных редакторов в Интернете довольно много. Вот один из них: McAfee FileWatch v1.0. Этот же редактор можно использовать для работы с исходными текстами программ. Однако мне больше нравится делать это с помощью следующего редактора:
7. **Текстовый редактор**. Необходим для написания исходных текстов ваших программ. Могу порекомендовать бесплатный редактор PSPad, который поддерживает множество языков программирования, в том числе и язык Ассемблера.

Все представленные в этой книге программы (и примеры программ) проверены на работоспособность. И именно эти программы используются для реализации примеров программ, приведённых в данной книге.

И еще – исходный код, написанный, например для Emu8086, будет немного отличаться от кода, написанного, например, для TASM. Эти отличия будут оговорены.

Большая часть программ, приведённых в книге, написана для MASM. Во-первых, потому что этот ассемблер наиболее популярен и до сих пор поддерживается. Во-вторых, потому что он поставляется с MSDN и с пакетом программ Visual Studio от Microsoft. Ну и в третьих, потому что я являюсь счастливым обладателем лицензионной копии MASM.

Если же у вас уже есть какой-либо ассемблер, не вошедший в перечисленный выше список, то вам придётся самостоятельно разобраться с его синтаксисом и почитать руководство пользователя, чтобы научиться правильно с ним работать. Но общие рекомендации, приведённые в данной книге, будут справедливы для любых (ну или почти для любых) ассемблеров.

Немного о процессорах

Процессор – это мозг компьютера. Физически это специальная микросхема с несколькими сотнями выводов, которая вставляется в материнскую плату. Если вы с трудом представляете себе, что это такое, рекомендую ознакомиться со статьёй Чайникам о компьютерах.

Процессоров существует довольно много даже в мире компьютеров. Но кроме компьютеров ещё есть телевизоры, стиральные машины, кондиционеры, системы управления двигателями внутреннего сгорания и т.п., где также очень широко используются процессоры (микропроцессоры, микроконтроллеры).

Каждый процессор обладает своим набором регистров. Регистры процессора – это такие специальные ячейки памяти, которые находятся непосредственно в микросхеме процессора. Регистры используются для разных целей (более подробно о регистрах будет написано ниже).

Каждый процессор имеет свой набор команд. Команда процессора записывается в определённый регистр, и тогда процессор выполняет эту команду. О командах процессора и регистрах мы будем говорить много и часто на протяжении всей книги. Для начинающих рекомендую книгу Как стать программистом, где в самых общих чертах, но зато понятным языком рассказано о принципах выполнения программы компьютером.

Что такое команда с точки зрения процессора? Это просто число. Однако современные процессоры могут иметь несколько сотен команд. Запомнить все их будет сложно. Как же тогда писать программы? Для упрощения работы программиста был придуман **язык Ассемблера**, где каждой команде соответствует мнемонический код. Например, число **4** соответствует мнемонике **ADD**. Иногда язык ассемблера ещё называют языком мнемонических команд.

1. БЫСТРЫЙ СТАРТ

1.1. Первая программа

Обычно в качестве первого примера приводят программу, которая выводит на экран строку «Hello World!». Однако для человека, который только начал изучать Ассемблер, такая программа будет слишком сложной (вы будете смеяться, но это действительно так – особенно в условиях отсутствия доходчивой информации). Поэтому наша первая программа будет еще проще – мы выведем на экран только один символ – английскую букву «A». И вообще – если вы уж решили стать программистом – срочно установите по умолчанию английскую раскладку клавиатуры. Тем более что некоторые ассемблеры и компиляторы не воспринимают русские буквы. Итак, наша первая программа будет выводить на экран английскую букву «A». Далее мы рассмотрим создание такой программы с использованием различных ассемблеров.

1.1.1. Emu8086

Если вы скачали и установили эмулятор процессора 8086 (см. раздел «ВВЕДЕНИЕ»), то вы можете использовать его для создания ваших первых программ на языке ассемблера. На текущий момент (ноябрь 2011 г) доступна версия программы 4.08. Справку на русском языке вы можете найти здесь: http://www.avprog.narod.ru/progs/emu8086/help.html.

Программа Emu8086 платная. Однако в течение 30 дней вы можете использовать её для ознакомления бесплатно.

Итак, вы скачали и установили программу Emu8086 на свой компьютер. Запускаем её и создаём новый файл через меню FILE – NEW – COM TEMPLATE (Файл – Новый – Шаблон файла COM). В редакторе исходного кода после этого мы увидим следующее:

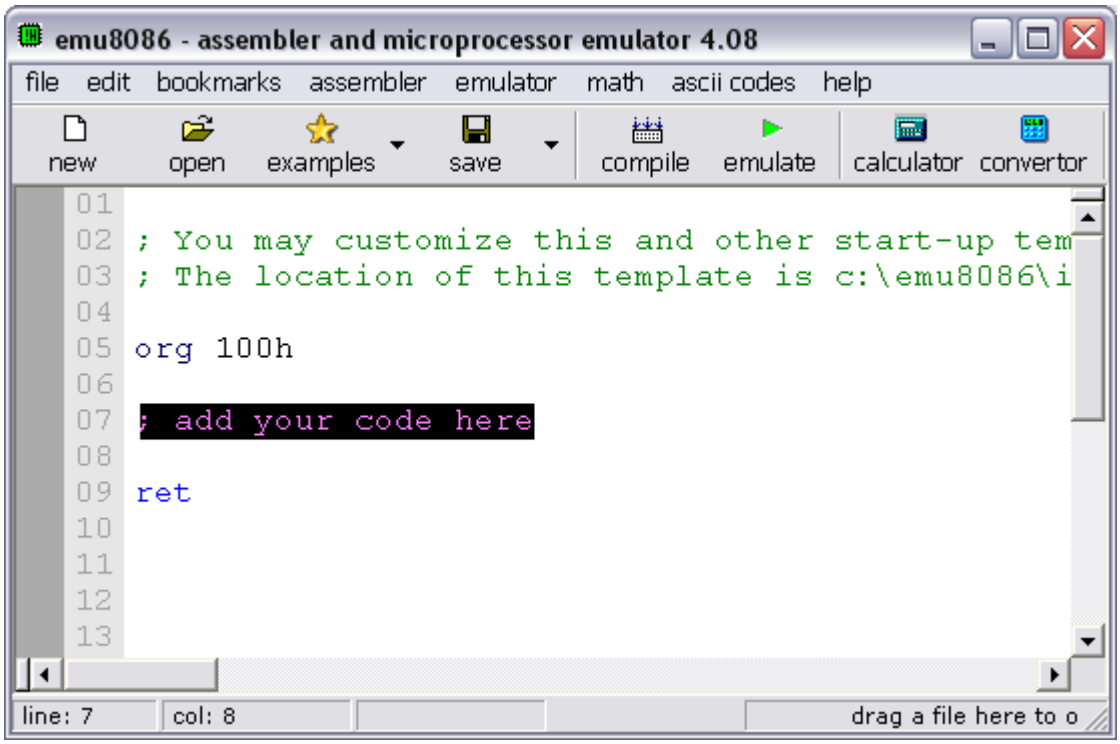

Рис. 1.1. Создание нового файла в Emu8086.

Здесь надо отметить, что программы, создаваемые с помощью Ассемблеров для компьютеров под управлением Windows, бывают двух типов: COM и EXE. Отличия между этими файлами мы рассмотрим позже, а пока вам достаточно знать, что на первое время мы будем создавать исполняемые файлы с расширением COM, так как они более простые.

После создания файла в Emu8086 описанным выше способом в редакторе исходного кода вы увидите строку «add your code hear» - «добавьте ваш код здесь» (рис. 1.1). Эту строку мы удаляем и вставляем вместо неё следующий текст:

```
MOV AH, 02h
MOV DL, 41h
INT 21h
INT 20h
```

Таким образом, полный текст программы будет выглядеть так:

```
ORG 100h

MOV AH, 02h
MOV DL, 41h
INT 21h
INT 20h

RET
```

Кроме этого в верхней части ещё имеются комментарии (на рис. 1.1 – это текст зелёного цвета). Комментарий в языке Ассемблера начинается с символа **;** (точка с запятой) и продолжается до конца строки. Если вы не знаете, что такое комментарии и зачем они нужны, см. книгу Как стать программистом. Как я уже говорил, здесь мы не будем растолковать азы программирования, так как книга, которую вы сейчас читаете, рассчитана на людей, знакомых с основами программирования.

Также отметим, что регистр символов в языке ассемблера роли не играет. Вы можете написать **RET**, **ret** или **Ret** – это будет одна и та же команда.

Вы можете сохранить этот файл куда-нибудь на диск. Но можете и не сохранять. Чтобы выполнить программу, нажмите кнопку EMULATE (с зелёным треугольником) или клавишу F5. Откроется два окна: окно эмулятора и окно исходного кода (рис. 1.2).

В окне эмулятора отображаются регистры и находятся кнопки управления программой. В окне исходного кода отображается исходный текст вашей программы, где подсвечивается строка, которая выполняется в данный момент. Всё это очень удобно для изучения и отладки программ. Но нам это пока не надо.

В окне эмулятора вы можете запустить вашу программу на выполнение целиком (кнопка RUN) либо в пошаговом режиме (кнопка SINGLE STEP). Пошаговый режим удобен для отладки. Ну а мы сейчас запустим программу на выполнение кнопкой RUN. После этого (если вы не сделали ошибок в тексте программы) вы увидите сообщение о завершении программы (рис. 1.3). Здесь вам сообщают о том, что программа передала управление операционной системе, то есть программа была успешно завершена. Нажмите кнопку OK в этом окне и вы увидите, наконец, результат работы вашей первой программы на языке ассемблера (рис. 1.4).

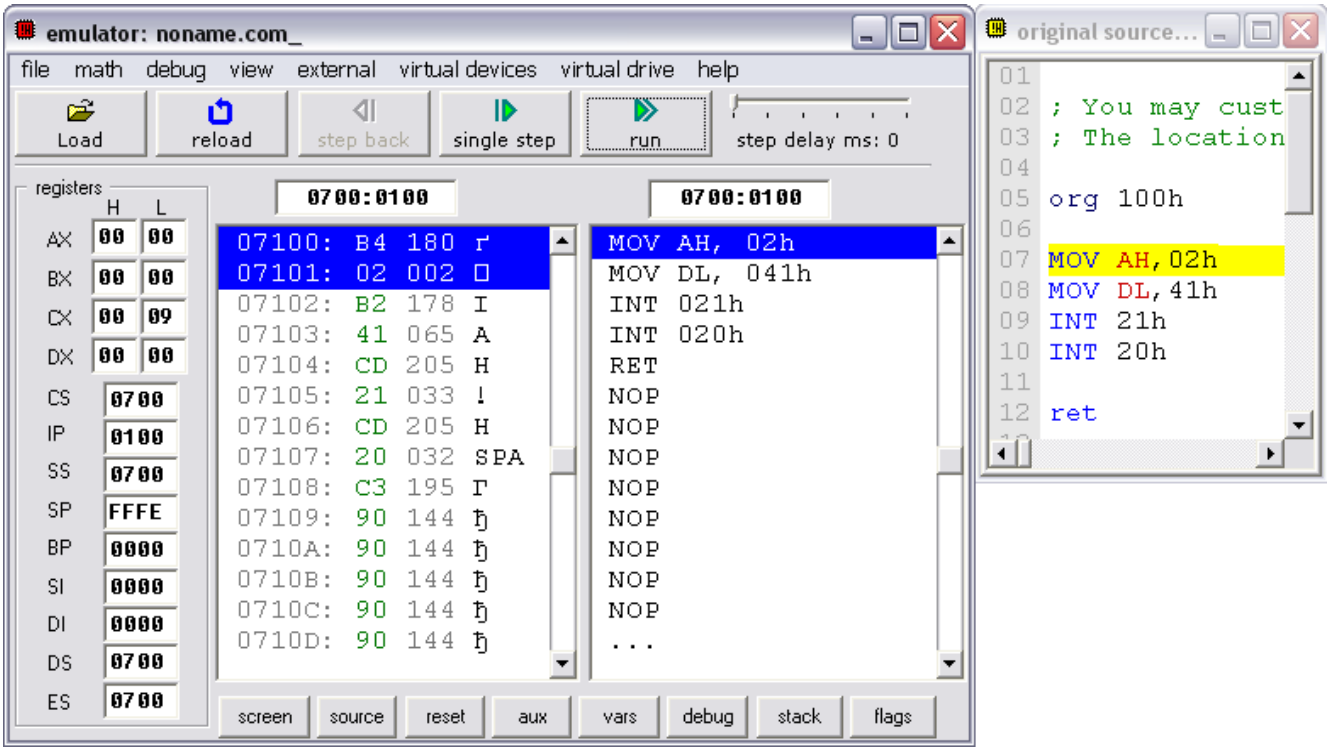

Рис. 1.2. Окно эмулятора Emu8086.

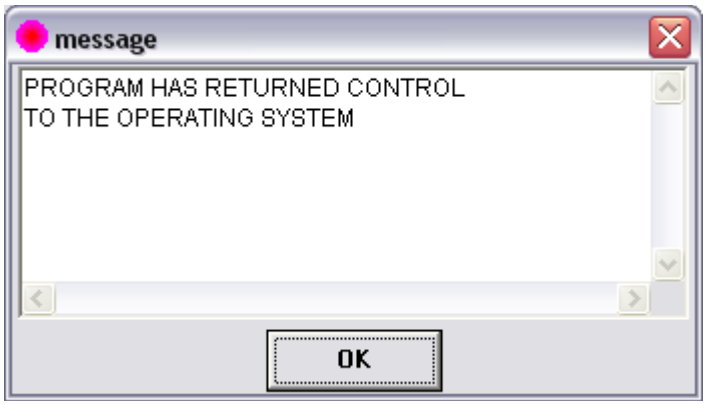

Рис. 1.3. Сообщение о завершении программы.

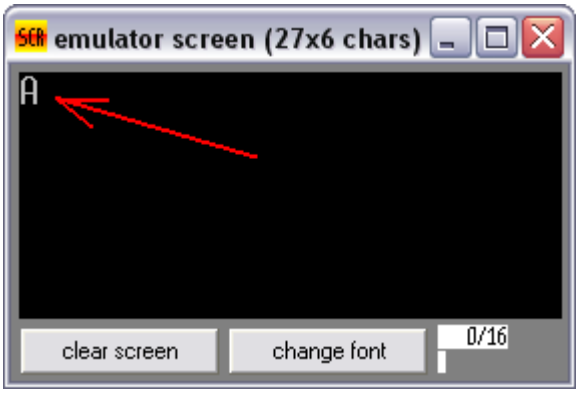

Рис. 1.4. Ваша первая программа выполнена.

Как мы уже говорили, наша первая программа выводит на экран английскую букву «A». Результат оправдал наши ожидания – буква «A» выведена на экран.

Здесь стоит отметить, что Emu8086 – это ЭМУЛЯТОР, то есть он эмулирует работу компьютера с процессором 8086. Поэтому в описанном выше примере программа выполняется не операционной системой, а эмулятором. Emu8086 может создавать и реальные программы, которые могут самостоятельно выполняться на компьютере. Но описание работы с Emu8086 не входит в наши планы. Читайте справку и экспериментируйте – всё у вас получится.

В нашем случае пока не важно, как выполняется программа – эмулятором или операционной системой. Главное – разобраться с вопросом создания программ на языке ассемблера. Поэтому разберём нашу простенькую программку подробно.

#make_COM# – 1-ая строка. Эта директива – специфическая для Emu8086. Она используется для определения типа создаваемого файла. В нашем случае это файл с расширением .COM.

ORG 100h – 2-ая строка. Эта команда устанавливает значение программного счетчика в 100h, потому что при загрузке COM-файла в память, DOS выделяет под блок данных PSP первые 256 байт (десятичное число 256 равно шестнадцатеричному 100). Код программы располагается только после этого блока. Все программы, которые компилируются в файлы типа COM, должны начинаться с этой директивы.

MOV AH, 02h – 3-я строка. Инструкция (или команда) **MOV** помещает значение второго операнда в первый операнд. То есть значение **02h** помещается в регистр **AH**. Для чего это делается? **02h** – это ДОСовская функция, которая выводит символ на экран. Мы пишем программу для DOS, поэтому используем команды этой операционной системы (ОС). А записываем мы эту функцию (а точнее ее номер) именно в регистр **AH**, потому что прерывание **21h** использует именно этот регистр.

MOV DL, 41h – 4-я строка. Код символа «A» заносится в регистр **DL**. Код символа «A» по стандарту ASCII – это **41h**.

INT 21h – 5-я строка. Это и есть то самое прерывание **21h** – команда, которая вызывает системную функцию DOS, заданную в регистре **AH** (в нашем примере это функция **02h**). Команда **INT 21h** – основное средство взаимодействия программ с ОС.

INT 20h – 6-я строка. Это прерывание, которое сообщает операционной системе о выходе из программы и о передаче управления консольному приложению. Значит, при использовании **INT 20h** в нашем примере, управление будет передаваться программе Emu8086. А в том случае, если программа уже откомпилирована и запущена из ОС, то команда **INT 20h** вернет нас в ОС (например, в DOS). В принципе, в случае с Emu8086 эту команду можно было бы пропустить, так как эту же функцию выполняет команда **RET**, которая вставляется в исходный текст автоматически при создании нового файла по шаблону (как это сделали мы ранее). Но я решил использовать **INT 20h** и здесь для совместимости с другими ассемблерами.

Тем, кому не все понятно из этих объяснений, рекомендую почитать книгу Как стать программистом, а также следующие главы.

1.1.2. Debug

Как уже говорилось (см. ВВЕДЕНИЕ), программа **Debug** входит в состав Windows. Запустить программу **Debug** можно из командной строки или непосредственно из папки, в которой она находится. Чтобы запустить программу из командной строки, выберите команду из меню ПУСК – ВЫПОЛНИТЬ или нажмите комбинацию клавиш WIN + R (если вы не знаете, что такое комбинация клавиш, см. книгу Компьютер для чайников). В открывшемся окне (рис. 1.5) напечатайте слово **debug** и нажмите клавишу ENTER или щёлкните кнопку OK.

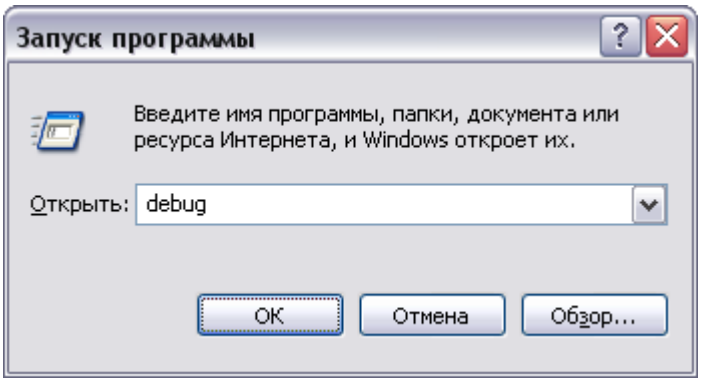

Рис. 1.5. Запуск программы DEBUG.

После этого откроется окно с пустым экраном и чёрточкой в левом верхнем углу, которая приглашает вас ввести какую-либо команду. Например, чтобы выйти из программы Debug, напечатайте букву **q** и нажмите ENTER.

Написать программу, используя Debug, можно не единственным способом, но мы пока рассмотрим тот, который больше похож на написание программы для Emu8086.

Чтобы написать уже известную нам программу, используя Debug, сделаем следующее (подразумевается, что Debug вы уже запустили, и увидели черный экран с маленькой мигающей черточкой в левом верхнем углу).

Введем букву «a» (напоминаю в последний раз - все команды вводятся на английском языке) и нажмем ENTER.

Затем введем программу, нажимая ENTER в конце каждой строки:

```
0B72: 0100 MOV AH, 02
0B72: 0102 MOV DL, 41
0B72: 0104 INT 21
0B72: 0106 INT 20
0B72: 0108
```

Результат будет примерно таким, как показано на рис. 1.6.

ПРИМЕЧАНИЕ 1:
Обратите внимание, что все числовые значения пишутся без буковки **h** в конце. Это потому, что Debug работает только с шестнадцатеричными числами, и ему не надо объяснять, в какой системе исчисления вводятся данные.

ПРИМЕЧАНИЕ 2:
После ввода команды **-a**, появляются символы: **0B72: 0100**. В вашем случае первые четыре символа могут быть другими, но нас они пока не интересуют. А как вы думаете, что означает число 0100? Помните директиву ORG 100h (см. раздел «1.1.1. Emu8086»)? Вот-вот – это адрес, с которого начинается выполнение программы. То есть в память с этим адресом заносится первая команда программы (для файлов COM). Каждая команда занимает 2 байта, поэтому следующий адрес будет 0102 и т.д.

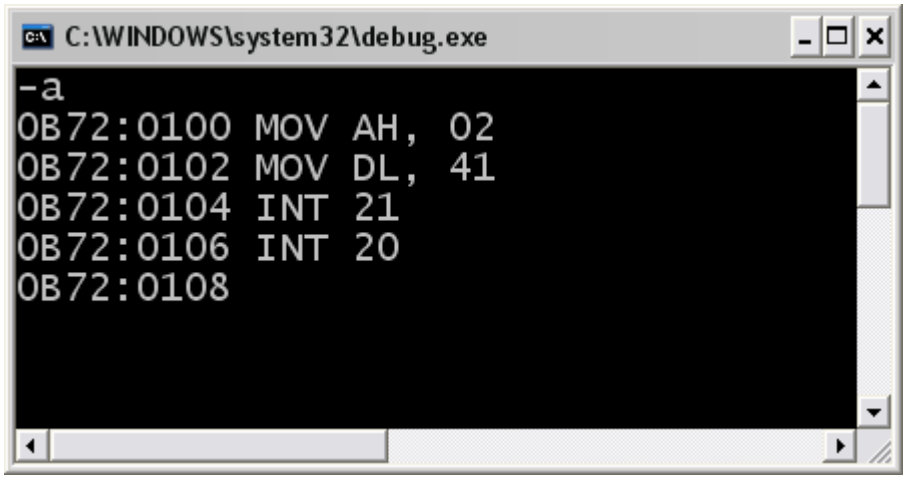

Рис. 1.6. Создание программы в Debug.

Сами команды мы уже знаем (см. раздел «1.1.1. Emu8086»), поэтому описывать их здесь не будем.

Программа написана – нужно проверить ее работу. Нажмём ENTER ещё раз, чтобы на экране появилась чёрточка, которая говорит о том, что можно вводить команду для Debug. Затем введем команду **g** (от английского «GO») и нажмем клавишу ENTER. На экране увидим следующее:

```
-g
A
Программа завершилась нормально
-
```

Здесь **A** – та самая буква, которая выводится на экран в результате работы программы. Затем идёт сообщение о нормальном завершении программы (оно может отличаться в зависимости от версии Debug).

А теперь, если интересно, можно проверить программу в пошаговом режиме, то есть понаблюдать, как выполняются команды одна за другой. Для этого придется по новой набрать текст программы (не забудьте сначала ввести команду **a**), затем:

Введем команду **t** и нажмем клавишу ENTER. Увидим нечто вроде этого:

```
AX=0200  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B72  ES=0B72  SS=0B72  CS=0B72  IP=0102  NV UP EI PL NZ NA PO NC
0B72:0102 B241          MOV   DL,41
```

Это есть ни что иное, как состояние регистров процессора после выполнения первой строки программы. Как вы можете видеть, в регистр AH записалось число 02. В нижней строке находится адрес команды и сама команда, которая будет выполняться следующей.

Снова введем команду **t** и нажмем клавишу ENTER. Увидим следующее:

```
AX=0200  BX=0000  CX=0000  DX=0041  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B72  ES=0B72  SS=0B72  CS=0B72  IP=0104  NV UP EI PL NZ NA PO NC
0B72:0104 CD21          INT   21
```

Команда **MOV DL, 41**, как ей и полагается, записала в регистр **DL** число **41**.

Снова введем команду **t** и нажмем клавишу ENTER. Увидим следующее:

```
AX=0200  BX=0000  CX=0000  DX=0041  SP=FFE8  BP=0000  SI=0000  DI=0000
DS=0B72  ES=0B72  SS=0B72  CS=0347  IP=0225  NV UP EI PL NZ NA PO NC
0347:0225 80FC4B  CMP   AH,4B
```

Команды **CMP AH,4B** нет в нашей программе. Наша программа завершила свою работу. Мы можем долго еще вводить команду **t** – нам будут выдаваться состояния регистров. Почему это происходит, нам пока не интересно. Лучше введем команду **g** и нажмем клавишу ENTER, таким образом окончательно выполним нашу программу, и увидим то, что мы уже видели.

Программа написана и проверена. Но как сделать ее самостоятельной, то есть как создать файл COM? Ведь то, что мы сделали, работает только с помощью Debug. Чтобы создать исполняемый файл, нужно ответить на несколько вопросов:

1. Какого размера будет наш файл? Выполнение программы начинается с адреса 0100h, а последняя строка в программе содержит адрес 0108h. Это значит, что размер файла будет 8 байт (108h – 100h = 8).
2. Как мы назовем наш файл? А хоть как. Однако, рекомендуется давать файлам английские имена, в которых содержится не более 8 символов (DOSy так приятнее работать). Назовем, например, debug_1.com

А теперь выполним следующие действия:

1. Снова напишем нашу программу (тренируйтесь, тренируйтесь...).
2. Запишем в регистр CX размер файла. Для этого введем команду **r cx** и нажмем ENTER. Затем введем размер файла (8 байт) и нажмем ENTER.
3. Введем команду **n**, затем один пробел и имя файла. Нажмем ENTER.
4. И, наконец, введем команду **w** и нажмем ENTER.

В результате всех этих действий на экране появится следующая информация (см. также рис. 1.7):

```
-r cx
CX 0000
:8
-n debug_1.com
-w
Запись: 00008 байт
-
```

Если вы работаете в режиме эмуляции DOS из под WINDOWS, то файл **debug_1.com** сохранится на рабочий стол, либо в папку текущего пользователя. Это зависит от версии и/или настроек WINDOWS. Теперь его можно запустить как обычную программу. Если в указанных папках вы не нашли этот файл, то найдите его через поиск файлов. Ну а если вы не знаете, как это сделать, см. книгу Компьютер для чайников.

Чувствую, что мы уже устали. Выход из Debug осуществляется командой **q**.

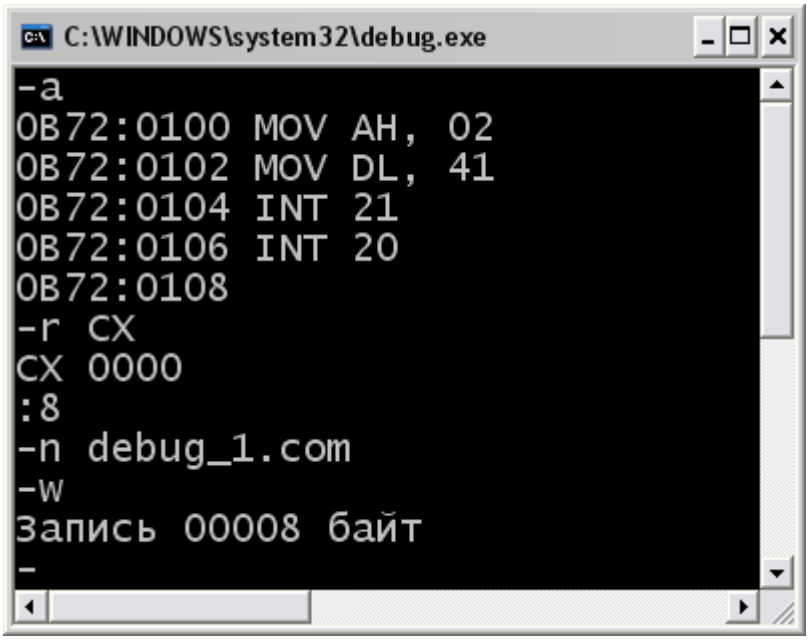

Рис. 1.7. Создание COM-файла с помощью Debug.

Набравшись сил и терпения, изучим еще одну опцию Debug – **дизассемблер**. С его помощью можно дизассемблировать какой-нибудь COM-файл (то есть выполнить действие, обратное ассемблированию – преобразовать исполняемый файл в исходный код на языке ассемблера). Допустим, у вас есть программка, написанная не вами – ее исходный код вы не знаете, а посмотреть очень хочется. Для этого и существует дизассемблер.

Итак, программа Debug у нас закрыта. Набираем в командной строке:

```
debug debug_1.com
```

(где **debug_1.com** – это имя файла, который мы хотим дизассемблировать) и нажимаем ENTER.

ПРИМЕЧАНИЕ:
Если программа не запустилась, значит нужно указать полный путь к ней, например

```
C:\WINDOWS\COMMAND\debug debug_1.com
```

Если же программа запустилась, но выдала ошибку (например: Ошибка 1282 или «Файл не найден»), то нужно указать полный путь к файлу, например:

```
C:\WINDOWS\COMMAND\debug C:\MYPROG\debug_1.com
```

Если и это не помогло, то, возможно, вы всё-таки где-то допустили ошибку в пути или путь не соответствует требованиям DOS. В таком случае лучше поместить программу в корень диска C, откуда она гарантированно загрузится по пути «C:\debug_1.com».

Если Debug запустилась без сообщений об ошибках, то вводим команду **u** и нажимаем ENTER. Вот что мы увидим (примерно, см. также рис 1.8):

```
-u
0BC6:0100 B402          MOV   AH, 02
0BC6:0102 B241          MOV   DL, 41
0BC6:0104 CD21          INT   21
0BC6:0106 CD20          INT   20
0BC6:0108 56            PUSH  SI
0BC6:0109 2E            CS:
0BC6:010A 8A04          MOV   AL, [SI]
0BC6:010C 0AC0          OR    AL, AL
0BC6:010E 741A          JZ    012A
0BC6:0110 3C3A          CMP   AL, 3A
0BC6:0112 750D          JNZ   0121
0BC6:0114 2E            CS:
0BC6:0115 807C0100      CMP   BYTE PTR [SI+01], 00
0BC6:0119 7506          JNZ   0121
0BC6:011B 2E            CS:
0BC6:011C C60400        MOV   BYTE PTR [SI], 00
0BC6:011F EB09          JMP   012A
-
```

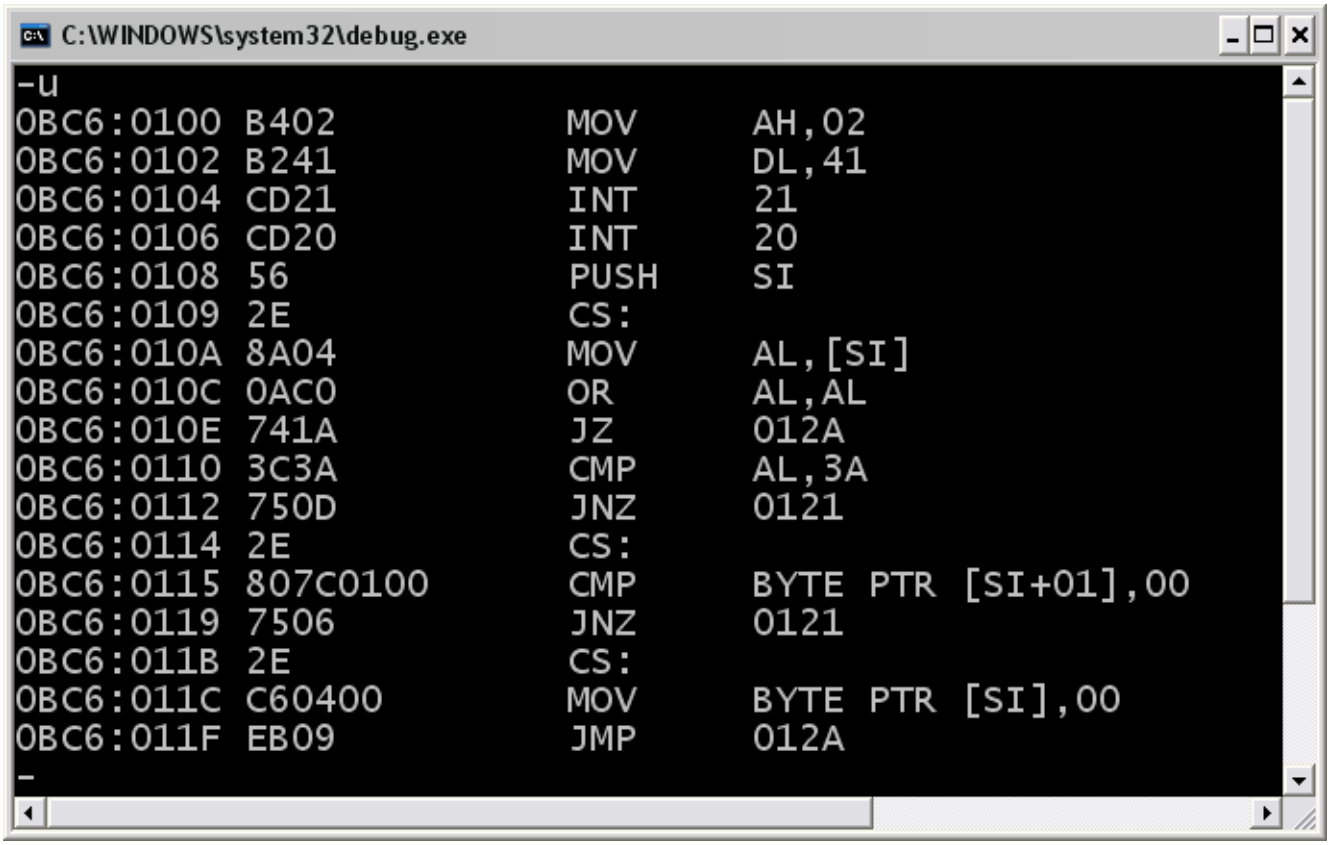

Рис. 1.8. Дизассемблер Debug.

Посмотрите на первые четыре строки. Узнаете? Это наша программа. Остальные строки нас не интересуют (это инструкции, оставшиеся от программ или данных, отработавших до запуска Debug). Ну а если мы рассматриваем незнакомый файл, как узнать, где кончается программа и начинается «мусор»? Ориентировочно это можно сделать по размеру файла (для очень маленьких программ). Размер можно посмотреть в свойствах файла. Только следует учитывать, что в свойствах файла размер дан в десятичной форме, а Debug нам выдает шестнадцатеричные адреса. Поэтому придется перевести десятичное число в шестнадцатеричное.

Есть еще вариант (который тоже не всегда приемлем) – найти в полученном списке строку, содержащую команду выхода из программы (INT 20).

Если программа большая, то список ее команд не поместится на экран. Тогда снова вводим команду **u** и нажимаем ENTER. И так до конца программы.

Возможно, вы не увидите на экране свою программу. Это может быть либо из-за того, что программа почему-то не загрузилась, либо по причине несоответствия адресов. Будьте внимательны: обращайте внимание на адреса памяти, которые указаны в левой колонке. Наша программа начинается с адреса 0100. Если адрес другой, то это, соответственно, не наша программа.

1.1.3. MASM, TASM и WASM

Ассемблеры MASM, TASM и WASM отличаются между собой. Однако создание простых программ для них практически не имеет отличий, за исключением самого ассемблирования и компоновки.

Итак, наша первая программа для MASM, TASM и WASM, которая выводит английскую букву «A» в текущей позиции курсора, то есть в левом верхнем углу экрана:

```
        .model tiny
        .code
        ORG     100h
start:  MOV     AH,2
        MOV     DL,41h
        INT     21h
        INT     20h
        END     start
```

Этот текст можно набрать в любом простом текстовом редакторе – например в БЛОКНОТЕ (NotePad) от WINDOWS (но не в Word и не в другом «навороченном»). Однако я рекомендую «продвинутый» текстовый редактор с подсветкой синтаксиса, например, PSPad (см. раздел ВВЕДЕНИЕ). Затем сохраняем этот файл с расширением **.asm**, например, в папке MYPROG. Назовем файл **atest**. Итак, мы получили: **C:\MYPROG\atest.asm**.

ПРИМЕЧАНИЕ:
Обратите внимание, что в первой команде мы записали 2 вместо 02h. MASM, TASM и WASM, как и Emu8086, допускают такие «вольности». Хотя можно написать 02h – ошибки не будет.

Пояснения к программе:

.model tiny – 1-ая строка. Директива **.model** определяет модель памяти для конкретного типа файлов. В нашем случае это файл с расширением COM, поэтому выбираем модель **tiny**, в которой объединены сегменты кода, данных, и стека. Модель **tiny** предназначена для создания файлов типа COM.

.code – 2-ая строка. Эта директива начинает сегмент кода.

ORG 100h – 3-ая строка. Эта команда устанавливает значение программного счетчика в 100h, потому что при загрузке COM-файла в память, DOS выделяет под блок данных PSP первые 256 байт (десятичное число 256 равно шестнадцатеричному 100h). Код программы располагается только после этого блока. Все программы, которые компилируются в файлы типа COM, должны начинаться с этой директивы.

start: MOV AH, 02h – 4-я строка. Метка **start** располагается перед первой командой в программе и будет использоваться в директиве END, чтобы указать, с какой команды начинается программа. Инструкция MOV помещает значение второго операнда в первый операнд. То есть значение 02h помещается в регистр AH. Для чего это делается? 02h - это ДОСовская функция, которая выводит символ на экран. Мы пишем программу для DOS, поэтому используем команды этой операционной системы (ОС). А записываем мы эту функцию (а точнее ее номер) именно в регистр AH, потому что прерывание 21h использует именно этот регистр.

MOV DL, 41h – 5-я строка. Код символа «A» заносится в регистр DL. Код символа «A» по стандарту ASCII – это число 41h.

INT 21h – 6-я строка. Это и есть то самое прерывание 21h – команда, которая вызывает системную функцию DOS, заданную в регистре AH (в нашем примере это функция 02h). Команда INT 21h – основное средство взаимодействия программ с ОС.

INT 20h – 7-я строка. Это прерывание, которое сообщает операционной системе о выходе из программы, и о передаче управления консольному приложению. В том случае, если программа уже откомпилирована и запущена из ОС, команда INT 20h вернет нас в ОС (например, в DOS).

END start – 8-я строка. Директива END завершает программу, одновременно указывая, с какой метки должно начинаться ее выполнение.

Ну вот, программу мы написали. Но хотелось бы посмотреть, как она работает. Для этого нужно сначала вызвать ассемблер, чтобы скомпилировать ее в объектный файл, а затем вызвать компоновщик, который из объектного файла создаст исполняемый файл, то есть программу типа COM. Для разных ассемблеров придётся выполнять эти действия по разному.

1.1.3.1. Ассемблирование в TASM

Для TASM создаём объектный файл с именем **atest.obj**, набрав в командной строке следующую команду:

```
tasm atest.asm
```

ПРИМЕЧАНИЕ:
Если на вашем компьютере программа **tasm** находится не в корневом каталоге диска C:\ и в файл **autoexec.bat** не внесены соответствующие изменения, то следует указывать полный путь к этой программе. Это касается и файла **atest.asm**. В этом случае команда может выглядеть, например, так:

```
C:\TASM\BIN\tasm C:\MYPROG\atest.asm
```

Далее следует скомпоновать полученный файл **atest.obj** в исполняемый файл **atest.com**. Для этого набираем команду:

```
tlink /t /x atest.obj
```

или полный путь:

```
C:\TASM\BIN\tlink /t /x atest.obj
```

Обратите внимание, что для файла **atest.obj** не нужно указывать полный путь, так как он записывается в папку C:\TASM\BIN\. Туда же по умолчанию записывается и готовая к выполнению программа **atest.com**. Теперь можно ее запустить, и мы увидим то, что должны увидеть – букву «A» в левом верхнем углу экрана.

ПРИМЕЧАНИЕ:
Если вы работаете в режиме эмуляции DOS из под WINDOWS, то файлы **atest.obj** и **atest.com** по умолчанию сохранятся на рабочий стол или в папку пользователя. Это зависит от версии и/или настроек WINDOWS.

1.1.3.2. Ассемблирование в MASM

Для MASM действия аналогичны, только вызов ассемблера и компоновщика несколько отличается. Создание объектного файла:

```
C:\MASM611\ml /c atest.asm
```

Компоновка объектного файла:

```
C:\MASM611\BINR\link /TINY atest.obj,,NUL,,,
```

Обратите внимание, что ассемблер и компоновщик находятся в разных каталогах. Здесь мы вызываем 16-разрядный компоновщик, который создаёт программы для реального режима DOS. Это справедливо для версии MASM 6.11, для других версий процесс создания может несколько отличаться – см. справку для вашей версии.

Файл **atest.com** по умолчанию создаётся в том же каталоге, где находятся исходный и объектный файл программы.

1.1.3.3. Ассемблирование в WASM

Для WASM действия аналогичны, только вызов ассемблера и компоновщика несколько отличается. Создание объектного файла:

```
wasm atest.asm
```

Компоновка объектного файла:

```
wlink file atest.obj form DOS COM
```

1.1.3.4. Выполнение программы

Если у вас всё получилось, то у вас появилась программа **atest.com**. Теперь её можно запустить на выполнение. Делается это обычным способом для операционной системы. Для ДОС – из командной строки. Для Windows – двойным щелчком по значку программы или также из командной строки.

Однако если вы работаете в Windows, то вы не успеете увидеть результат работы программы, так как окно сразу же закроется после её выполнения. Чтобы этого не произошло, щёлкните правой кнопкой по файлу **atest.com** и в контекстном меню выберите СВОЙСТВА. В открывшемся окне перейдите на вкладку ПРОГРАММА и снимите галочку «Закрывать окно по завершении работы» (в зависимости от версии Windows эта процедура может немного отличаться).

Рис. 1.9. Наша первая программа.

После выполнения программы мы увидим на экране то, что и должны были увидеть – английскую букву **A** (рис. 1.9). Можете немного поэкспериментировать и вместо кода буквы «A» (41h) записать другой код (41h, 42h и т.п. вплоть до 0FFh).

1.1.3.5. Использование BAT-файлов

Как вы уже наверняка убедились, ассемблирование программ дело довольно скучное. Приходится набирать в командной строке довольно много букв. А если вы пишите реальную программу, то повторять эту операцию придётся очень много раз.

Существенно упростить эту процедуру можно с помощью старых добрых BAT-файлов. BAT-файл (или пакетный файл) – это обычный текстовый файл с расширением BAT, в котором записываются команды для выполнения операционной системой. Точно также, как вы это делаете в командной строке. Только в BAT-файле можно записать сразу несколько команд, и все эти команды затем можно выполнить щелчком мыши. Для любопытных рекомендую ознакомиться с контрольной работой по BAT-файлам, где приведены примеры создания относительно сложных файлов. Набравшись немного опыта, вы можете создать универсальный BAT-файл, который позволит вам быстро ассемблировать и компоновать ваши исходные тексты на языке ассемблера.

Но здесь мы создадим простейший BAT-файл, с помощью которого «лёгким движением руки» мы выполним ассемблирование и компоновку, и создадим исполняемый файл типа COM с помощью ассемблера MASM. Итак, откроем наш любимый текстовый редактор (у меня это PSPad, вы можете воспользоваться блокнотом). Создадим новый файл и напишем там следующий текст:

```
C:\MASM611\BIN\ml /c atest.asm
PAUSE
C:\MASM611\BINR\link /TINY atest.obj,,NUL,,,
PAUSE
```

Здесь команда PAUSE приостанавливает выполнение команд BAT-файла и выводит сообщение «Для продолжения нажмите ENTER…». Само собой, что команды продолжат выполняться после нажатия на ENTER.

Сохраним этот файл с расширением BAT в том же каталоге, где у нас находится исходный файл **atest.asm**. Назовём его, например, **com_create.bat**. В результате папка с исходными файлами в проводнике будет выглядеть примерно так, как показано на рис. 1.10.

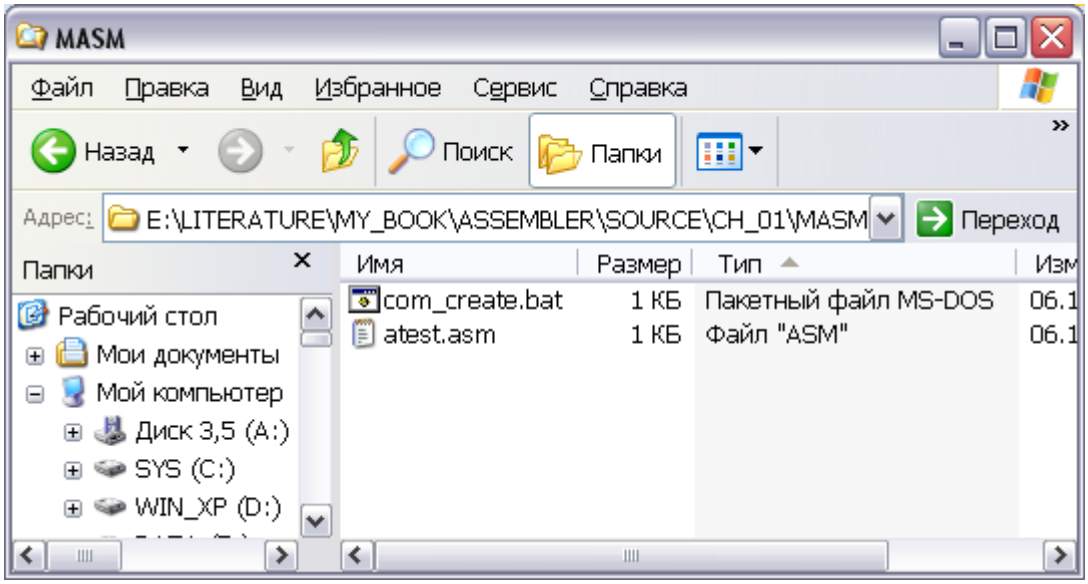

Рис. 1.10. Файл com_create.bat в ПРОВОДНИКЕ.

Если в вашем случае в графе «Тип» написано не «Пакетный файл MS-DOS», а что-то другое (например, текстовый файл), то это значит, что вы плохо представляете себе, что такое расширение файла. В этом случае настоятельно рекомендую ознакомиться с книгой Компьютер для чайников.

Теперь выполним этот BAT-файл обычным для Windows способом, то есть дважды щёлкнем по нему левой кнопкой мыши. Что же произойдёт? Операционная система начнёт поочерёдно выполнять команды, записанные в пакетном файле. Сначала выполнится ассемблирование (создание объектного файла). Затем выполнится команда PAUSE. Эта команда здесь для того, чтобы вы могли посмотреть результат ассемблирования. После нажатия клавиши ENTER выполнится компоновка (создание исполняемого файла типа COM, то есть создание готовой программы). Затем снова будет пауза, чтобы вы могли увидеть результат. На экране это будет выглядеть примерно так, как показано на рис. 1.11.

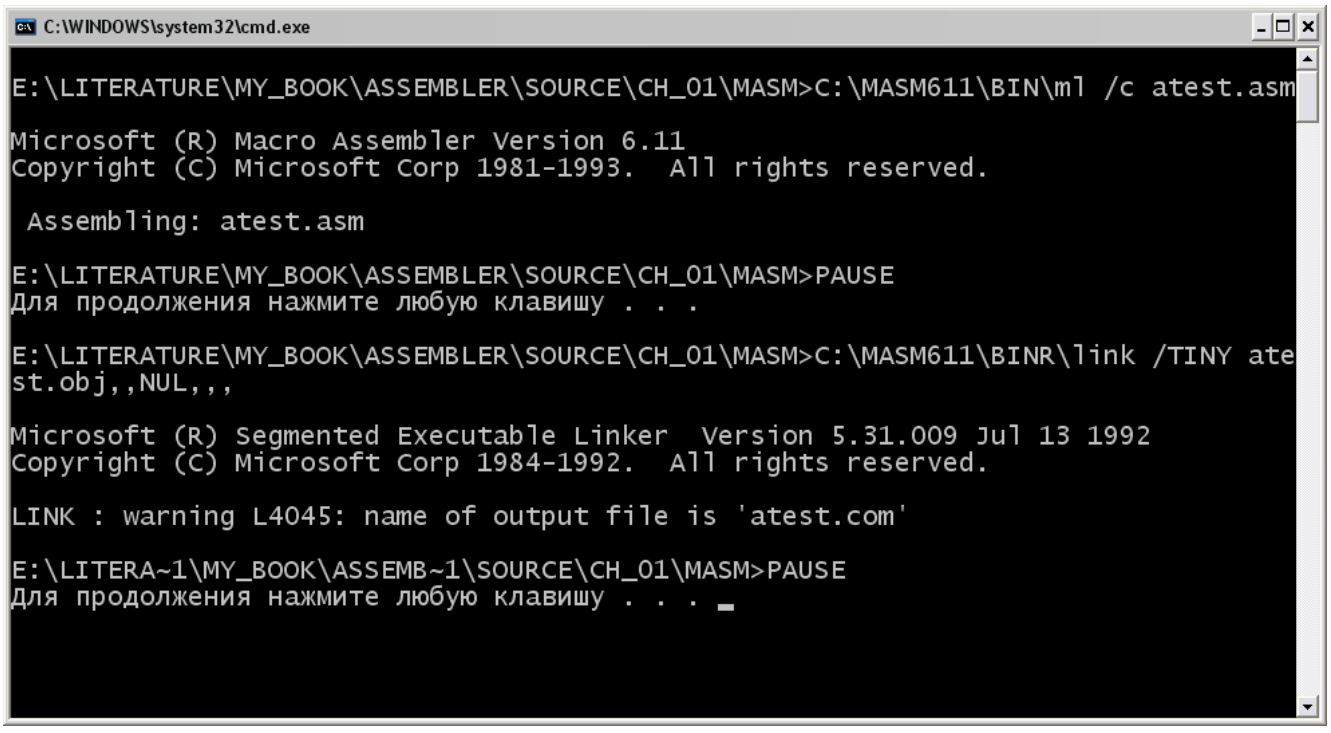

Рис. 1.11. Создание программы на MASM с помощью BAT-файла.

Конечно, пути в вашем случае будут другими. Как видим, сначала выполняется ассемблирование:

```
Assembling: attest.asm
```

Затем выполняется команда PAUSE:

```
ПутьКBATФайлу>PAUSE
```

После нажатия ENTER выполняется компоновка:

```
LINK: warning L4045: name of output file is 'atest.com'
```

Здесь нам сообщают, что компоновщик создал выходной файл **attest.com**. В чем мы и можем убедиться, заглянув снова в наш каталог с исходными файлами.

Как видите, ассемблирование и компоновка исходных кодов на ассемблере становится не таким уж сложным делом, если подойти к этому творчески. Созданный нами BAT-файл вы можете скопировать в другую папку с другими исходными кодами. Вам останется только заменить имя исходного файла (в тексте выше выделено красным) и файл будет готов к работе с другими исходными кодами.

1.1.4. Шестнадцатеричный редактор

То, что мы сделаем сейчас, с моей точки зрения весьма интересно. Это будет ваша первая **программа в машинных кодах** (и, скорее всего, единственная))).

Ассемблер – это язык низкого уровня, но все ж таки язык. А пробовали вы написать программу в машинных кодах? Сейчас попробуем.

Написать программу можно и не имея никаких ассемблеров-компиляторов и прочих инструментов – с помощью какого-либо шестнадцатеричного редактора.

ВНИМАНИЕ!
Написание программ с использованием шестнадцатеричного редактора – это не только утомительно, но и НЕБЕЗОПАСНО ДЛЯ КОМПЬЮТЕРА! Так как ошибки в процессе создания программы неизбежны. Но если TASM со товарищи проверяют текст программы на наличие ошибок, то в шестнадцатеричном редакторе проверяющий только один – вы сами. Поэтому, если ошибка останется незамеченной, файл все равно будет создан. И если вы попытаетесь этот «неправильный» файл запустить, то в лучшем случае получите зависание компьютера, а в худшем этот файл может такое натворить – вирусы отдыхают.

И все-таки разбор программ в шестнадцатеричном редакторе весьма полезен. Особенно тем, кто собирается работать с электроникой – ведь микропроцессоры не понимают ни Паскаль ни C++. Хотя и существуют специальные устройства и программы, которые им эти языки «объясняют».

Для начала вам потребуется шестнадцатеричный редактор. Вы можете использовать любой, имеющийся у вас под рукой. Однако я буду использовать уже упоминавшийся McAfee FileWatch v1.0. Все описанные ниже действия справедливы именно для этого редактора.

Итак, шестнадцатеричный редактор у вас установлен. Запускаем его. Щелкаем по кнопке ОТКРЫТЬ, находим один из созданных нами COM-файлов, например, **debug_1.com**, и загружаем его в редактор.

Когда файл загружен, в редакторе вы увидите следующее (см. также рис. 1.12):

```
00000000   B4  02  B2  41  CD  21  CD  20      ...A.!.
```

Можете открыть два других созданных нами файла: **mycode.com** (созданный в emu8086) или **ATEST.COM** (который мы создали в разделе «1.1.3. MASM, TASM и WASM»). Увидите то же самое. Это значит, что все ассемблеры создают одинаковый машинный код. То есть отличия в тексте программ не являются принципиальными – они обусловлены только отличиями самих ассемблеров.

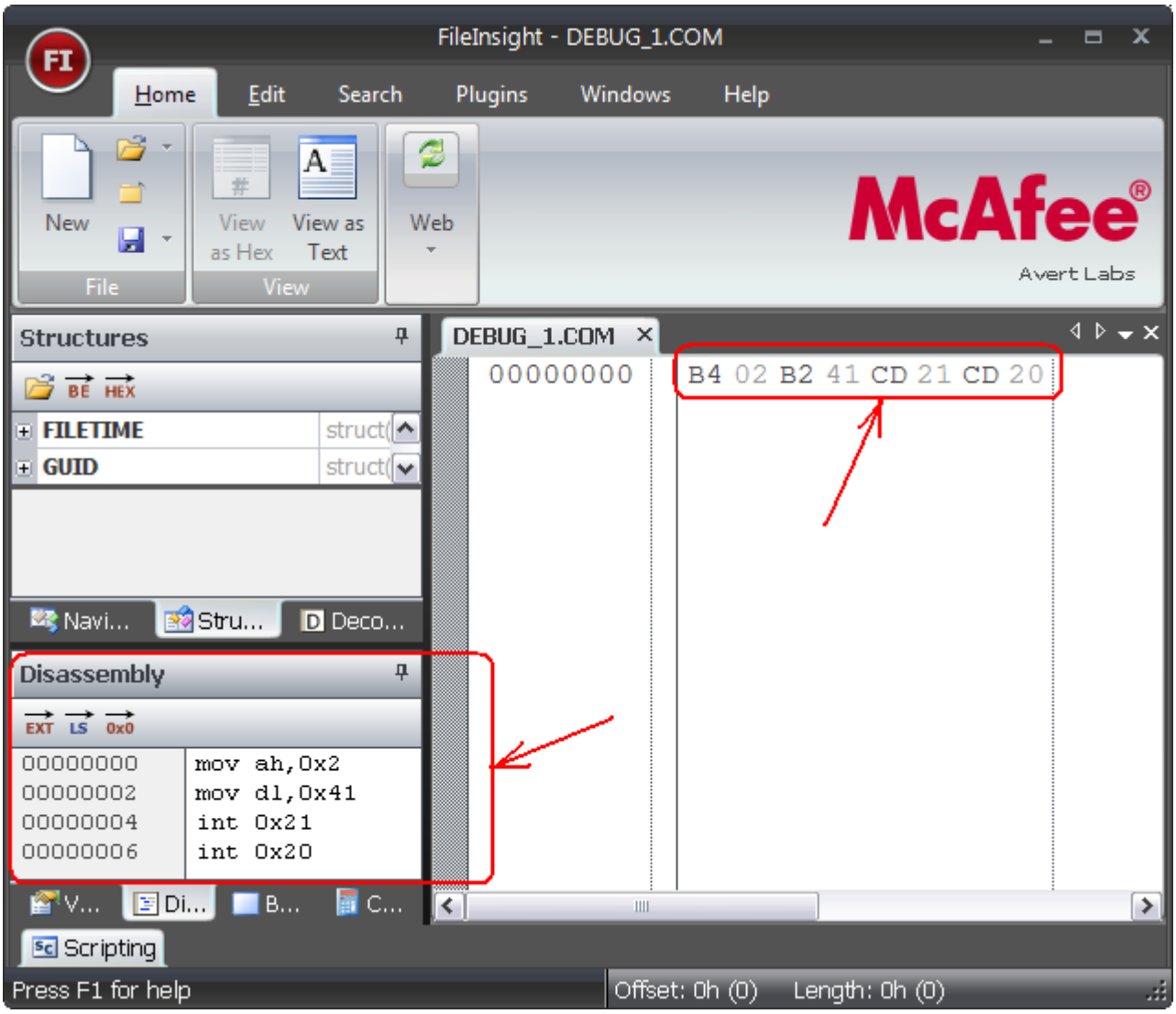

Рис. 1.12. Файл DEBUG_1.COM в шестнадцатеричном редакторе.

ПРИМЕЧАНИЕ
Если в вашем случае вы видите другую картину, то либо вы открыли другой файл, либо просматриваете его в текстовом режиме. В последнем случае нажмите кнопку **View as Hex** на панели инструментов (см. рис. 1.12).

Что же означают эти числа?

С нулями все понятно – это первая ячейка памяти, в которую записано число B4. Это число потом будет записано в адрес 0100h (для COM-файла). В строке должно быть 16 чисел, каждое из которых состоит из двух цифр. Числа записываются в шестнадцатеричной форме. Но у нас программа маленькая – всего 8 байт, поэтому и чисел у нас 8.

Ну а что же такое B4? Это команда – «Ввести значение в регистр AH». А какое значение вводим? Правильно: 02 (следующее в строке число).

Идем дальше. Команда B2 – «ввести значение в регистр DL». Какое значение? Конечно 41. А теперь вспомним, что мы увидели в программе Debug, введя команду **t**:

```
AX=0200  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B72  ES=0B72  SS=0B72  CS=0B72  IP=0102  NV UP EI PL NZ NA PO NC
0B72:0102 B241          MOV   DL,41
```

Видите в последней строке B241? Знакомое сочетание? Это код команды **MOV DL, 41**.

Идем дальше. CD – код вызова прерывания. Какого? Смотрим следующее число. Итак, CD 21 – это код команды INT 21. Аналогично для CD 20.

Осталось разобраться с загадочными символами в конце строки. А здесь все просто: каждая цифра в числе соответствует коду символа таблицы ASCII, и эти символы выводятся в той же последовательности, что и шестнадцатеричные цифры. В этом тексте вместо некоторых символов стоят точки (.) – это просто коды не буквенных символов.

Ну а теперь напишем и создадим нашу изученную вдоль и поперек программу без ассемблеров и компоновщиков. Открываем редактор, создаём новый файл (для этого щёлкаем кнопку NEW на панели инструментов), затем щёлкаем кнопку **View as Hex** и вводим данные:

```
00000000   B4  02  B2  41  CD  21  CD  20
```

Сохраняем файл под именем, например, **hex_1.com**. Все. Программа готова. Теперь ее можно запустить и в очередной раз полюбоваться своим творением. Результат будет тот же, что и во всех предыдущих случаях.

И ещё один приятный сюрприз от редактора McAfee FileWatch v1.0 – он имеет свой дизассемблер! Если вы загрузите в редактор исполняемый файл, а в левом нижнем углу выберите вкладку DISASSEMBLY, то сможете посмотреть исходный код загруженной программы на языке ассемблера (рис. 1.12).

Зачем вообще нужны шестнадцатеричные редакторы и дизассемблеры? Ведь это так сложно. Да, это непросто. Однако хакеры так не думают. Именно с помощью шестнадцатеричных редакторов и дизассемблеров они ломают программы. Находят в коде нужные им места и исправляют их в соответствии со своими хакерскими капризами.

Конечно, мы не хакеры. Ломать программы не будем. Однако дизассемблеры и шестнадцатеричные редакторы весьма полезны и законопослушными программистам. Они используются, например, для отладки, для изучения машинных кодов и т.п. Например, вы знаете, как выглядит команда на языке ассемблера, но хотите узнать её машинный код. Если нет документации, то выход только один – шестнадцатеричный редактор и/или дизассемблер. Следует, однако, учесть, что не все команды умещаются в машинный код из двух чисел. Некоторые команды довольно сложные и требуют большего количества чисел для представления в машинных кодах.

Резюме

В общих чертах мы ознакомились с языком Ассемблера и с разными методами создания программ на этом языке. Подведём некоторые итоги:

1. Язык Ассемблера – это набор мнемонических обозначений команд процессора. Каждый процессор имеет свой набор команд. Мнемоники команд, которые выполняют одинаковые функции, могут отличаться для разных процессоров и/или ассемблеров.
2. Для каждого процессора существует свой ассемблер. Для каждого ассемблера существует свой язык ассемблера, хотя для одинаковых процессоров языки ассемблера могут быть очень похожи.
3. Для каждого процессора существует свой набор регистров. Названия регистров также могут отличаться в зависимости от модели процессора.
4. BAT-файлы – очень полезная вещь. Многие системные администраторы и программисты до сих пор широко используют пакетные файлы, несмотря на то, что сейчас существует множество «скриптовых» языков, таких как VBScript.
5. Ещё одна полезная вещь – шестнадцатеричные редакторы и дизассемблеры. Если вы хотите всерьёз научиться программировать разное «железо», то без них вам не обойтись. Да и в программировании на языках высокого уровня временами возникает необходимость в этих очень полезных инструментах.

2. ВВЕДЕНИЕ В АССЕМБЛЕР

Если вы в полной мере удовлетворили своё любопытство, прочитав раздел Быстрый старт, значит душа ваше не предрасположена к программированию на языке Ассемблера. В этом случае, быть может, вам стоит попробовать программирование на языках высокого уровня или вообще сменить профессию/хобби.

Для тех же, кто только раззадорился и жаждет продолжить изучение ассемблера, предназначены остальные разделы данной книги. Сразу скажу, что вам придётся изучить и понять множество различных материалов, которые, на первый взгляд, непосредственно программирования не касаются, однако без которых профессиональное программирование невозможно. Это очень долгий и трудный путь. Либо вы его достойно пройдёте и станете авторитетным специалистом, либо так и останетесь любителем, который нахватался верхушек и даже может писать разные программки, но также отличается от профессионала, как самолёт от воздушного змея. И тот и другой может летать, но самолёт – это серьёзная и сложная машина, а воздушный змей – это детская игрушка.

Но хватит лирики. Пора начать разговор по теме. В этом разделе мы рассмотрим не только основы программирования на языке ассемблера, но и такие необходимые вещи, как системы счисления, устройство процессора и компьютера в целом, организацию памяти и многое другое. Всё это крайне необходимо знать и понимать, потому что программируя на языке ассемблера, вы будете напрямую работать с регистрами процессора, с памятью и железом. Для вычисления математических выражений вам нужно чётко понимать, как различные числа представлены в памяти компьютера. Также вам нужно знать, как вообще работает процессор (и компьютер), иначе вы не сможете чувствовать себя уверенно при написании программ на языке ассемблера.

Ещё раз предлагаю вам ознакомиться с книгой Как стать программистом, где в самых общих чертах описано устройство компьютера и процессора. Тогда вам легче будет понять материалы, изложенные далее в этом разделе.

В данной книге мы будем говорить только о программировании для 16-разрядной ОС DOS. Однако это не должно вас смущать. Во-первых, существуют эмуляторы, где вы можете в полной мере протестировать ваши программы. Во-вторых, программы, написанные для DOS, в большинстве случаев будут нормально работать под Windows. В-третьих, в большинстве случаев программы для Windows лучше создавать с помощь современных визуальных средств разработки, таких как Delphi или Visual Basic. Ну и, в-четвёртых, в будущих изданиях этой книги я обязательно добавлю разделы по программированию для Windows.

2.1. Как устроен компьютер

Если вы опытный пользователь, а тем более программист, то вы знаете, что основными элементами компьютера являются системный блок, монитор и клавиатура. Но это с точки зрения пользователя. Поскольку вы взялись за изучение ассемблера, то вам пора мыслить как профессионалу. А вот с точки зрения профессионала простейший компьютер содержит следующие элементы:

1. Процессор
2. Оперативная память (ОЗУ)
3. Устройства ввода-вывода
4. Шина данных
5. Шина адреса
6. Шина управления

В принципе этого достаточно для решения большинства задач.

Оперативная память предназначена для загрузки программ и для временного хранения различных данных, необходимых для работы программ.

Устройства ввода-вывода предназначены для взаимодействия с пользователем и другими устройствами. Например, монитор предназначен для вывода информации пользователю. Клавиатура предназначена для получения информации от пользователя, то есть для ввода информации.

Конечно, существуют и другие устройства ввода-вывода, например, мышь. Однако с точки зрения программиста на языке Ассемблера тип устройства ввода-вывода особого значения не имеет, так как работа с такими устройствами ведётся через порты ввода-вывода. Поэтому достаточно знать тип информации и номер порта ввода-вывода. Но об этом позже.

Шина (bus) – это группа параллельных проводников, с помощью которых данные передаются от одного устройства к другому. Обычно компьютер состоит из трёх шин:

- **Шина данных** (data bus) используется для обмена команд и данных между процессором и оперативной памятью, а также между устройствами ввода-вывода и ОЗУ.
- **Шина управления** (control bus) используется для передачи специальных сигналов, которые синхронизируют работу всех устройств, подключенных к системной шине. Например, процессор должен знать, когда можно читать информацию с шины данных. Для этого используется специальный сигнал готовности шины данных.
- **Шина адреса** (address bus) используется для указания адреса ячейки памяти в ОЗУ, к которой в текущий момент происходит обращение со стороны процессора или устройства ввода-вывода (чтение или запись).

Самый основной элемент компьютера, это, конечно, процессор. Об устройстве процессора мы будем говорить в следующем разделе.

2.1.1. Структура процессора

Упрощённая структура процессора показана на рис. 2.1.

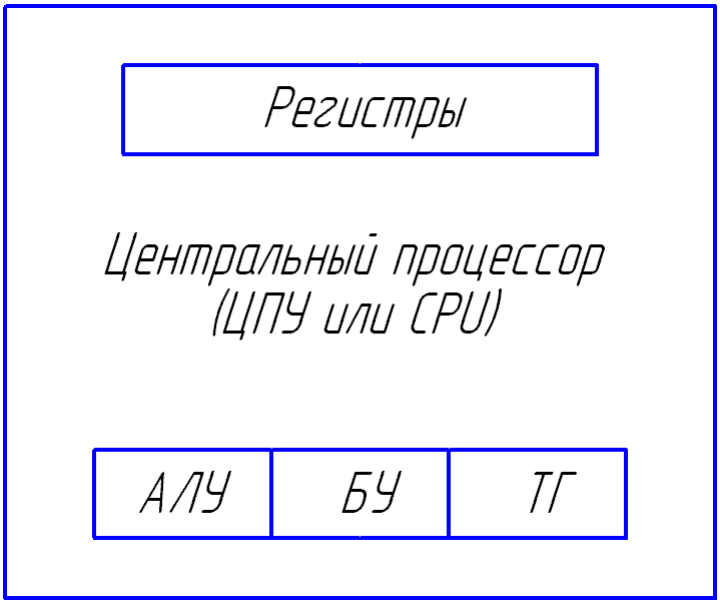

Рис. 2.1. Структура процессора.

Основные элементы процессора:

- Регистры
- АЛУ – арифметико-логическое устройство
- БУ – блок управления
- ТГ – тактовый генератор

Регистры – это специальные ячейки памяти, физически расположенные внутри процессора. В отличие от ОЗУ, где для обращения к данным требуется использовать шину адреса, к регистрам процессор может обращаться напрямую. Это существенно ускорят работу с данными.

Арифметико-логическое устройство выполняет арифметические операции, такие как сложение, вычитание, а также логические операции.

Блок управления определяет последовательность микрокоманд, выполняемых при обработке машинных кодов (команд).

Тактовый генератор, или **генератор тактовых импульсов**, задаёт рабочую частоту процессора. С помощью тактовых импульсов выполняется синхронизация для внутренних команд процессора и остальных устройств. Тактовый генератор вырабатывает (генерирует) прямоугольные импульсы, которые следуют с определённой частотой (для разных процессоров частота разная).

В теории электронно-вычислительных машин различают два понятия: машинный такт и машинный цикл.

Машинный такт соответствует одному периоду импульсов тактового генератора и является основной единицей измерения времени выполнения команд процессором.

Машинный цикл состоит из нескольких машинных тактов. Машинный цикл – это время, необходимое для выполнения одной команды.

Машинный цикл может отличаться для разных команд. Для простых команд может потребоваться всего 1-2 машинных такта. В то время как для сложных команд, таких как умножение, может потребоваться до 50 машинных тактов и более. Это очень важный момент. Когда вы будете писать реальные программы, которые очень критичны к быстродействию, следует помнить о том, что разные команды требуют соответствующего времени работы процессора. То есть одни и те же действия можно выполнить, например, за 100 машинных тактов, а можно и за 20. Это зависит от опыта и квалификации программиста, а также от конкретных задач.

Доработка программы таким образом, чтобы она выполнялась максимально быстро (то есть для её выполнения требовалось как можно меньше машинных тактов) называется **оптимизация по быстродействию**. В таких случаях часто приходится чем-то жертвовать, например, усложнять программу или увеличивать её размер. Есть и другие типы оптимизации, например, **оптимизация по размеру**. В этом случае обычно жертвуют быстродействием, чтобы получить программу с минимальным размером исполняемого файла. Выбор оптимизации зависит от конкретной задачи. Вопросы оптимизации будут рассмотрены в соответствующем разделе.

2.1.2. Регистры процессора

Начиная с модели 80386 процессоры Intel предоставляют 16 основных регистров для пользовательских программ и ещё 11 регистров для работы с мультимедийными приложениями (MMX) и числами с плавающей точкой (FPU/NPX). Все команды так или иначе изменяют содержимое регистров. Как уже говорилось, обращаться к регистрам быстрее и удобнее, чем к памяти. Поэтому при программировании на языке Ассемблера регистры используются очень широко.

В этом разделе мы рассмотрим основные регистры процессоров Intel. Названия и состав/количество регистров для других процессоров могут отличаться. Итак, основные регистры процессоров Intel.

Таблица 2.1. Основные регистры процессора.

Название	Разрядность	Основное назначение
EAX	32	Аккумулятор
EBX	32	База
ECX	32	Счётчик
EDX	32	Регистр данных
EBP	32	Указатель базы
ESP	32	Указатель стека
ESI	32	Индекс источника
EDI	32	Индекс приёмника
EFLAGS	32	Регистр флагов
EIP	32	Указатель инструкции (команды)
CS	16	Сегментные регистры
DS	16	
ES	16	
FS	16	
GS	16	
SS	16	

Регистры EAX, EBX, ECX, EDX – это регистры общего назначения. Они имеют определённое назначение (так уж сложилось исторически), однако в них можно хранить любую информацию.

Регистры EBP, ESP, ESI, EDI – это также регистры общего назначения. Они имеют уже более конкретное назначение. В них также можно хранить пользовательские данные, но делать это нужно уже более осторожно, чтобы не получить «неожиданный» результат.

Регистр флагов и сегментные регистры требуют отдельного описания и будут более подробно рассмотрены далее.

Пока для вас здесь слишком много непонятных слов, но со временем всё прояснится)))

Когда-то процессоры были 16-разрядными, и, соответственно, все их регистры были также 16-разрядными. Для совместимости со старыми программами, а также для удобства программирования некоторые регистры разделены на 2 или 4 «маленьких» регистра, у каждого из которых есть свои имена. В таблице 2.2 перечислены такие регистры.

Вот пример такого регистра.

Разряд (бит)	Регистр (32 бита)	Регистр (16 бит)	Регистр (8 бит)
31	EAX	Старшие разряды регистра EAX	
30			
29			
28			
27			
26			
25			
24			
23			
22			
21			
20			
19			
18			
17			
16			
15		AX	AH
14			
13			
12			
11			
10			
9			
8			
7			AL
6			
5			
4			
3			
2			
1			
0			

Из этого следует, что вы можете написать в своей программе, например, такие команды:

```
MOV   AX,  1
MOV   EAX, 1
```

Обе команды поместят в регистр AX число 1. Разница будет заключаться только в том, что вторая команда обнулит старшие разряды регистра EAX, то есть после выполнения второй команды в регистре EAX будет число 1. А первая команда оставит в старших разрядах регистра EAX старые данные. И если там были данные, отличные от нуля, то после выполнения первой команды в регистре EAX будет какое-то число, но не 1. А вот в регистре AX будет число 1. Сложно? Ну это пока… Со временем вы к таким вещам привыкните.

Мы пока не говорили о разрядах (битах). Эту тему мы обсудим в разделах, посвящённых системам счисления. А сейчас пока вам достаточно знать, что нулевой разряд (бит) – это младший бит. Он крайний справа. Старший бит – крайний слева. Номер старшего бита зависит от разрядности числа/регистра. Например, в 32-разрядном регистре старшим битом является 31-й бит (потому что отсчёт начинается с 0, а не с 1).

Ниже приведён список регистров общего назначения, которые можно поделить описанным выше способом и при этом к «половинкам» и «четвертинкам» этих регистров можно обращаться в программе как к отдельному регистру.

Таблица 2.2. «Делимые» регистры.

Регистр	Старшие разряды	Имена 16-ти и 8-ми битных регистров	
	31…16	15…8	7…0
EAX		AX	
		AH	AL
EBX		BX	
		BH	BL
ECX		CX	
		CH	CL
EDX		DX	
		DH	DL
ESI		SI	
EDI		DI	
EBP		BP	
ESP		SP	
EIP		IP	

На этом мы закончим наше краткое знакомство с регистрами. Если вам пока не всё понятно – просто прочитайте этот раздел, чтобы более-менее представлять себе, что такое регистры. По мере приобретения новых знаний вы можете вернуться к этому разделу и уже на новом уровне воспринять эту информацию. А в следующем разделе мы коротко опишем процесс выполнения команды.

2.1.3. Цикл выполнения команды

Программа состоит из машинных команд. Программа загружается в оперативную память компьютера. Затем программа начинает выполняться, то есть процессор выполняет машинные команды в той последовательности, в какой они записаны в программе.

Для того чтобы процессор знал, какую команду нужно выполнять в определённый момент, существует **счётчик команд** – специальный регистр, в котором хранится адрес команды, которая должна быть выполнена после выполнения текущей команды. То есть при запуске программы в этом регистре хранится адрес первой команды. В процессорах Intel в качестве счётчика команд (его ещё называют **указатель команды**) используется регистр EIP (или IP в 16-разрядных программах).

Счётчик команд работает со сверхоперативной памятью, которая находится внутри процессора. Эта память носит название **очередь команд**, куда помещается одна или несколько команд непосредственно перед их выполнением. То есть в счётчике команд хранится адрес команды в очереди команд, а не адрес оперативной памяти.

Цикл выполнения команды – это последовательность действий, которая совершается процессором при выполнении одной машинной команды. При выполнении каждой машинной команды процессор должен выполнить как минимум три действия: выборку, декодирование и выполнение. Если в команде используется операнд, расположенный в оперативной памяти, то процессору придётся выполнить ещё две операции: выборку операнда из памяти и запись результата в память. Ниже описаны эти пять операций.

- **Выборка команды**. Блок управления извлекает команду из памяти (из очереди команд), копирует её во внутреннюю память процессора и увеличивает значение счётчика команд на длину этой команды (разные команды могут иметь разный размер).
- **Декодирование команды**. Блок управления определяет тип выполняемой команды, пересылает указанные в ней операнды в АЛУ и генерирует электрические сигналы управления АЛУ, которые соответствуют типу выполняемой операции.
- **Выборка операндов**. Если в команде используется операнд, расположенный в оперативной памяти, то блок управления начинает операцию по его выборке из памяти.
- **Выполнение команды**. АЛУ выполняет указанную в команде операцию, сохраняет полученный результат в заданном месте и обновляет состояние флагов, по значению которых программа может судить о результате выполнения команды.
- **Запись результата в память**. Если результат выполнения команды должен быть сохранён в памяти, блок управления начинает операцию сохранения данных в памяти.

Суммируем полученные знания и составим цикл выполнения команды:

1. Выбрать из очереди команд команду, на которую указывает счётчик команд.
2. Определить адрес следующей команды в очереди команд и записать адрес следующей команды в счётчик команд.
3. Декодировать команду.
4. Если в команде есть операнды, находящиеся в памяти, то выбрать операнды.
5. Выполнить команду и установить флаги.
6. Записать результат в память (по необходимости).
7. Начать выполнение следующей команды с п.1.

Это упрощённый цикл выполнения команды. К тому же действия могут отличаться в зависимости от процессора. Однако это даёт общее представление о том, как процессор выполняет одну машинную команду, а значит и программу в целом.

2.1.4. Организация памяти

С точки зрения процессора память – это последовательность байтов, каждому из которых присвоен уникальный адрес со значениями от 0 до (2^{32} – 1), то есть до 4 ГБ. Конечно, сейчас есть 64-разрядные процессоры. Но о них в этой книге мы говорить не будем.

Программы могут работать с памятью как с одним непрерывным массивом (модель памяти flat – плоская) или как с несколькими массивами (сегментированные модели памяти). Во втором случае для задания адреса любого байта требуется два числа – адрес начала массива и адрес байта внутри этого массива.

Кроме основной памяти программы могут использовать регистры процессора, о которых говорилось выше.

Выбор метода обращения к памяти определяется режимом работы процессора. Процессоры Intel могут работать в одном из трёх основных режимах:

- Реальный режим (режим реальной адресации – Real-address mode)
- Защищённый режим (Protected mode)
- Режим управления системой (System Management mode)

Более подробно о режимах процессора мы поговорим как-нибудь в другой раз. А сейчас нас интересуют различия при работе с памятью в зависимости от режима.

В **реальном режиме** процессор может обращаться только к первому мегабайту памяти, адреса которого находятся в диапазоне 00000…FFFFF. При этом процессор работает в однопрограммном режиме, то есть одновременно может выполняться только одна программа. Реальный режим работы используется в операционной системе DOS, а также в системах Windows 95/98 при загрузке в режиме эмуляции DOS.

В **защищённом режиме** процессор может одновременно выполнять несколько программ. При этом каждой программе может быть назначено до 4 ГБ оперативной памяти. Чтобы предотвратить влияние программ друг на друга, им выделяются изолированные участки памяти. Поэтому режим и называется защищённым. В защищённом режиме работают такие системы как Windows и Linux.

Об организации памяти в реальном и защищённом режимах мы поговорим в следующих разделах. А пока рассуждения о памяти закончим. Тема эта большая и для кого-то может оказаться сложной. К ней мы ещё будем возвращаться. Некоторую информацию о сегментированных моделях памяти можно найти здесь: Контрольная работа по информатике.

2.1.5. Реальный режим

В реальном режиме процессор может обращаться только к первым 1 048 576 байтам (1 МБ) оперативной памяти, так как в реальном режиме используется только 20 младших разрядов шины адреса. Из этого следует, что диапазон адресов памяти (в шестнадцатеричном представлении) будет от 00000 до FFFFF. А повелось это с тех времён, когда шина адреса была 20-разрядной, а регистры 16-разрядными. То есть одного регистра было недостаточно для хранения адреса.

В реальном режиме используется сегментная модель памяти. Суть её заключается в том, что всё доступное адресное пространство разделено на блоки по 64 КБ. Такие блоки называются **сегментами**.

Пример записи адреса в сегментной модели памяти:

```
8000:0100
```

Здесь 8000 – это адрес сегмента (или просто **сегмент**), а 0100 – это смещение относительно адреса сегмента (или просто **смещение**). Таким образом, чтобы получить доступ к какому-либо байту в памяти в реальном режиме, нужно знать сегмент и смещение, то есть начало 64-килобайтного блока памяти, где находится нужный нам байт, и смещение от начала этого блока, то есть адрес (номер) байта в этом блоке. Напомню, что нумерация начинается с нуля, а не с единицы.

Реальный адрес (линейный адрес) нужного нам байта определяется следующим образом (упрощено в расчёте на начинающих, профессионалов прошу отнестись с пониманием). Берём сегмент и приписываем к нему нолик справа, то есть в нашем примере получим 80000. Затем прибавляем к полученному числу смещение. Получаем 80100 – это и есть линейный адрес, то есть адрес, который используется в 20-разрядной шине адреса для доступа к байту. Операцию преобразования сегментного адреса в линейный адрес выполняет **сумматор адреса** – специальное аппаратное устройство, которое входит в состав процессора.

Напомню, что все адреса в данном разделе представлены в шестнадцатеричной системе.

Ещё немного наглядной информации о сегментных и линейных адресах вы можете найти здесь: Контрольная работа по информатике.

2.1.6. Защищённый режим

При работе в защищённом режиме каждой программе может быть выделен блок памяти размером до 4 ГБ. Адреса этого блока в шестнадцатеричном представлении могут меняться от 00000000 до FFFFFFFF. В защищённом режиме программе выделяется **линейное адресное пространство** (flat address space), которое разработчики компилятора Microsoft Assembler назвали **линейная модель памяти** (flat memory model) или **плоская модель памяти**.

С точки зрения программиста линейная модель памяти более проста в использовании, так как для хранения адреса любой переменной или команды достаточно одного 32-разрядного регистра.

Ну что же. На первый раз информации об организации памяти достаточно. Подозреваю, что даже эту информацию многие читатели просмотрели «по диагонали». Это объяснимо – тема довольно сложная, если вы сталкиваетесь с этим в первый раз. Рекомендую приступить к изучению следующих разделов, а к памяти мы ещё вернёмся…

Продолжение следует…

Обновления ищите здесь:
http://av-assembler.ru/asm/afd/assembler-for-dummy.htm

ПРОГРАММИРОВАНИЕ В MASM

Обзор MASM