Олег Калашников

Учимся программировать

2-е издание

Основные команды процессоров Intel

16- и 32-разрядные регистры

Основы работы с сопроцессором

Управление XMS-памятью

Разработка и написание резидентных программ, файловой оболочки, вируса и антивируса

Исследование работы отладчиков, принципы отладки программ сторонних авторов

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1
К17

Калашников О. А.

К17 Ассемблер — это просто. Учимся программировать. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2011. — 336 с.: ил. + CD-ROM

ISBN 978-5-9775-0591-8

Подробно и доходчиво объясняются все основные вопросы программирования на ассемблере. Рассмотрены команды процессоров Intel, 16- и 32-разрядные регистры, основы работы с сопроцессором, сегментация памяти в реальном масштабе времени, управление клавиатурой и последовательным портом, работа с дисками и многое другое. Описано, как разработать безобидный нерезидентный вирус и антивирус против этого вируса, как написать файловую оболочку (типа Norton Commander или FAR Manager) и как писать резидентные программы.

Каждая глава состоит из объяснения новой темы, описания алгоритмов программ, многочисленных примеров и ответов на часто задаваемые вопросы. Во второе издание внесены исправления и добавлены новые примеры. Компакт-диск содержит исходные коды всех примеров, приведенных в книге, с подробными описаниями.

Для программистов

УДК 681.3.068+800.92Ассемблер
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

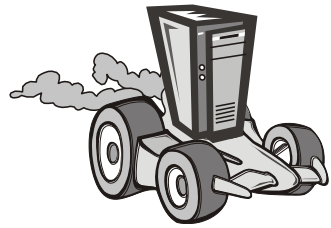
Лицензия ИД № 02429 от 24.07.00. Подписано в печать 31.01.11.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 27,09.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12.

ISBN 978-5-9775-0591-8

Оглавление



Глава 1

Первая программа

1.1. Шестнадцатеричная система счисления

Для написания программ на ассемблере необходимо разобраться с шестнадцатеричной системой счисления. Ничего сложного в ней нет. Мы используем в жизни десятичную систему. Не сомневаемся, что вы с ней знакомы, поэтому попробуем объяснить шестнадцатеричную систему, проводя аналогию с десятичной.

Итак, в десятичной системе, если мы к какому-нибудь числу справа добавим ноль, то это число увеличится в 10 раз. Например:

$1 \times 10 = 10$

$10 \times 10 = 100$

$100 \times 10 = 1000$

и т. д.

В этой системе мы используем цифры от 0 до 9, т. е. десять разных цифр (собственно, поэтому она и называется десятичной).

В шестнадцатеричной системе мы используем, соответственно, шестнадцать "цифр". Слово "цифр" специально написано в кавычках, т. к. в этой системе используются не только цифры. От 0 до 9 мы считаем так же, как и в десятичной, а вот дальше таким образом: A, B, C, D, E, F. Число F, как не трудно посчитать, будет равно 15 в десятичной системе (табл. 1.1).

Таблица 1.1. *Десятичная и шестнадцатеричная системы*

Десятичное число	**Шестнадцатеричное число**	**Десятичное число**	**Шестнадцатеричное число**
0	0	26	1A
1	1	27	1B
2	2	28	1C
3	3	29	1D
4	4	30	1E
...	...	...	...
8	8	158	9E

Таблица 1.1 *(окончание)*

Десятичное число	Шестнадцатеричное число	Десятичное число	Шестнадцатеричное число
9	9	159	9F
10	A	160	A0
11	B	161	A1
12	C	162	A2
13	D	...	...
14	E	254	FE
15	F	255	FF
16	10	256	100
17	11	257	101
...	...	...	...

Таким образом, если мы к какому-нибудь числу в шестнадцатеричной системе добавим справа ноль, то это число увеличится в 16 раз (пример 1.1).

Пример 1.1

```
1 x 16 = 10
10 x 16 = 100
100 x 16 = 1000
```

и т. д.

Вы смогли отличить в примере 1.1 шестнадцатеричные числа от десятичных? А из этого ряда: 10, 12, 45, 64, 12, 8, 19? Это могут быть как шестнадцатеричные числа, так и десятичные. Для того чтобы не было путаницы, а компьютер и программист смогли бы однозначно отличить одни числа от других, в ассемблере принято после шестнадцатеричного числа ставить символ h или H (от англ. *hexadecimal* — шестнадцатеричное), который для краткости часто называют просто hex. После десятичного числа, как правило, ничего не ставят. Так как числа от 0 до 9 в обеих системах имеют одинаковые значения, то числа, записанные как 5 и 5h, — одно и то же. Таким образом, корректная запись чисел из примера 1 будет следующей (примеры 1.2 и 1.3).

Пример 1.2. Корректная форма записи чисел

```
1 x 16 = 10h
10h x 16 = 100h
100h x 16 = 1000h
```

Пример 1.3. Другой вариант записи чисел

```
1h x 10h = 10h
10h x 10h = 100h
100h x 10h = 1000h
```

Для чего нужна шестнадцатеричная система и в каких случаях она применяется — мы рассмотрим в следующих главах. А в данный момент для нашего примера программы, который будет рассмотрен далее, нам необходимо знать о существовании шестнадцатеричных чисел.

Итак, настала пора подвести промежуточный итог. Шестнадцатеричная система счисления состоит из 10 цифр (от 0 до 9) и 6 букв латинского алфавита (A, B, C, D, E, F). Если к какому-нибудь числу в шестнадцатеричной системе добавить справа ноль, то это число увеличится в 16 раз. Очень важно уяснить принцип шестнадцатеричной системы счисления, т. к. мы будем постоянно использовать ее при написании наших программ на ассемблере.

Теперь немного о том, как будут строиться примеры на ассемблере в данной книге. Не совсем удобно приводить их сплошным текстом, поэтому сперва будет идти сам код программы с пронумерованными строками, а сразу же после него — объяснения и примечания. Примерно так, как показано в листинге 1.1.

Листинг 1.1. Пример записи ассемблерных инструкций, применяемой в книге

```
...
(01)    mov ah,9
(02)    mov al,8
...
(15)    mov dl,5Ah
...
```

Обратите внимание, что номера строк ставятся только в книге, и при наборе программ в текстовом редакторе эти номера ставить НЕ нужно! Номера строк ставятся для того, чтобы удобно было давать объяснения к каждой строке: в строке (01) мы делаем то-то, а в строке (15) — то-то.

Несмотря на то, что на компакт-диске, прилагаемом к книге, имеются набранные и готовые для ассемблирования программы, мы рекомендуем все-таки первое время набирать их самостоятельно. Это ускорит запоминание операторов, а также облегчит привыкание к самому языку.

И еще момент. Строчные и ПРОПИСНЫЕ символы программой-ассемблером не различаются. Записи вида:

```
mov ah,9
```

и

```
MOV AH,9
```

воспринимаются одинаково. Можно, конечно, заставить ассемблер различать регистр, но мы пока этого делать не будем. Для удобства чтения программы лучше всего операторы вводить строчными буквами, а названия подпрограмм и меток начинать с прописной.

1.2. Наша первая программа

Итак, переходим к нашей первой программе (\001\prog01.asm) (листинг 1.2).

Листинг 1.2. Наша первая программа на ассемблере

```
(01) CSEG segment
(02) org 100h
(03)
(04) Begin:
(05)
(06)    mov ah,9
(07)    mov dx,offset Message
(08)    int 21h
(09)
(10)    int 20h
(11)
(12) Message db 'Hello, world!$'
(13) CSEG ends
(14) end Begin
```

Еще раз обратим внимание: когда вы будете перепечатывать примеры программ, то номера строк ставить не нужно!

В скобках указывается имя файла из архива файлов-приложений (в данном случае — \001\prog01.asm, где 001 — каталог, prog01.asm — имя ассемблерного файла в DOS-формате).

Прежде чем пытаться ассемблировать, прочтите данную главу до конца!

Для того чтобы объяснить все операторы из листинга 1.2, нам потребуется несколько глав. Поэтому описание некоторых команд мы на данном этапе опустим. Просто считайте, что так должно быть. В ближайшее время мы рассмотрим эти операторы подробно. Итак, строки с номерами (01), (02) и (13) вы игнорируете. Строки (03), (05), (09) и (11) остаются пустыми. Это делается для наглядности и удобства программиста при просмотре и анализе кода. Программа-ассемблер пустые строки опускает.

Теперь перейдем к рассмотрению остальных операторов. Со строки (04) начинается код программы. Это метка, указывающая ассемблеру на начало кода. В строке (14) стоят операторы `end Begin` (`Begin` — начало; `end` — конец). Это конец программы. Вообще вместо слова `Begin` можно было бы использовать любое

другое. Например, `Start`. В таком случае, нам пришлось бы и завершать программу оператором `End Start` (14).

Строки (06)—(08) выводят на экран сообщение "Hello, world!". Здесь придется вкратце рассказать о регистрах процессора (более подробно эту тему мы рассмотрим в последующих главах).

Регистр процессора — это специально отведенная память для хранения какого-нибудь числа. Например, если мы хотим сложить два числа, то в математике запишем так:

A = 5
B = 8
C = *A* + *B*

A, *B* и *C* — это своего рода регистры (если говорить о компьютере), в которых могут храниться некоторые данные. *A* = 5 следует читать как: *"присваиваем A число 5"*.

Для присвоения регистру какого-нибудь значения в ассемблере существует оператор `mov` (от англ. *move* — в данном случае "загрузить"). Строку (06) следует читать так: *"загружаем в регистр* `ah` *число 9"* (проще говоря, присваиваем `ah` число 9). Далее рассмотрим, зачем это необходимо. В строке (07) загружаем в регистр `dx` адрес сообщения для вывода (в данном примере это будет строка `"Hello, world!$"`). Затем, в строке (08) вызываем прерывание MS-DOS, которое и выведет нашу строку на экран. Прерывания будут подробно рассматриваться в последующих главах, мы же пока коснемся только самых элементарных вещей.

1.3. Введение в прерывания

Прерывание MS-DOS — это своего рода подпрограмма (часть MS-DOS), которая находится постоянно в памяти и может вызываться в любое время из любой программы. Рассмотрим вышесказанное на примере (листинг 1.3).

Сразу стоит отметить, что в ассемблере после точки с запятой располагаются *комментарии*. Комментарии будут опускаться MASM/TASM при ассемблировании. Примеры комментариев:

```
;это комментарий
mov ah,9    ;это комментарий
```

В комментарии программист вставляет замечания по программе, которые помогают сориентироваться в коде.

Листинг 1.3. Программа (алгоритм) сложения двух чисел

```
НачалоПрограммы
A=5           ;в переменную A заносим значение 5
B=8           ;в переменную B значение 8
ВызовПодпрограммы Addition
;теперь C равно 13

A=10         ;то же самое, только другие числа
```

```
B=25
ВызовПодпрограммы Addition
;теперь C равно 35

КонецПрограммы
;выходим из программы

...
Подпрограмма Addition
C = A + B
ВозвратИзПодпрограммы
;возвращаемся в то место, откуда вызывали
КонецПодпрограммы
```

В данном примере мы дважды вызвали подпрограмму (процедуру) `Addition`, которая произвела сложение двух чисел, переданных ей в переменных `A` и `B`. Результат математического действия сохраняется в переменной `C`. Когда вызывается подпрограмма, компьютер запоминает, с какого места она была вызвана, и после того, как процедура отработала, возвращается в то место, откуда она вызывалась. Таким образом, можно вызывать подпрограммы неопределенное количество раз с любого участка основной программы.

При выполнении строки (08) (см. листинг 1.2) мы вызываем подпрограмму (в данном случае это называется прерыванием), которая выводит на экран строку. Для этого мы, собственно, и помещаем нужные значения в регистры, т. е. готовим для прерывания необходимые параметры. Всю работу (вывод строки, перемещение курсора) берет на себя эта процедура. Строку (08) следует читать так: *"вызываем двадцать первое прерывание"* (`int` от англ. *interrupt* — прерывание). Обратите внимание, что после числа 21 стоит буква h. Это, как мы уже знаем, шестнадцатеричное число (33 в десятичной системе). Конечно, нам ничего не мешает заменить строку `int 21h` строкой `int 33`. Программа будет работать корректно. Но в ассемблере принято указывать номера прерываний в шестнадцатеричной системе, да и все отладчики работают с этой системой.

В строке (10) мы, как вы уже догадались, вызываем прерывание `20h`. Для его вызова не нужно указывать какие-либо значения в регистрах. Оно выполняет только одну задачу — выход из программы (выход в DOS). В результате выполнения прерывания `20h` программа вернется туда, откуда ее запускали (загружали, вызывали). Например, в Norton Commander или DOS Navigator. Это что-то вроде оператора `exit` в некоторых языках высокого уровня.

Строка (12) содержит сообщение для вывода. Первое слово (`message` — сообщение) — название этого сообщения. Оно может быть любым (например, `mess` или `string` и пр.). Обратите внимание на строку (07), в которой мы загружаем в регистр `dx` адрес этого сообщения.

Можно создать еще одну строку, которую назовем `Mess2`. Затем, начиная со строки (09), вставим в нашу программу следующие команды:

```
...
(09)     mov ah,9
(10)     mov dx,offset Mess2
(11)     int 21h
(12)     int 20h
(13) Message db 'Hello, world!$'
(14) Mess2 db 'Это Я!$'
(15) CSEG ends
(16) end Begin
```

Уверены, вы поняли, что именно произойдет.

Обратите внимание на последний символ в строках `Message` и `Mess2` — `$`. Он указывает на конец выводимой строки. Если мы его уберем, то прерывание `21h` продолжит вывод до тех пор, пока не встретится где-нибудь в памяти тот самый символ `$`. На экране, помимо нашей строки, мы увидим "мусор" — разные символы, которых в строке вовсе нет.

Теперь ассемблируйте программу. Как это сделать — написано в *приложении 1*. Заметьте, что мы создаем пока только COM-файлы, а не EXE! Для того чтобы получить COM-файл, нужно указать определенные параметры ассемблеру (MASM/TASM) в командной строке. Пример получения COM-файла с помощью Macro Assembler версии 6.11 и результат выполнения программы приведен на рис. 1.1. При возникновении ошибок в процессе ассемблирования обращайтесь к *приложению 2*, где рассматриваются типичные ошибки при ассемблировании программ.

Если у вас есть отладчик (AFD, CodeView), то можно (и даже нужно!) запустить эту программу под его управлением. Это поможет вам лучше понять структуру и принцип работы ассемблера, а также продемонстрирует реальную работу написанной нами программы.

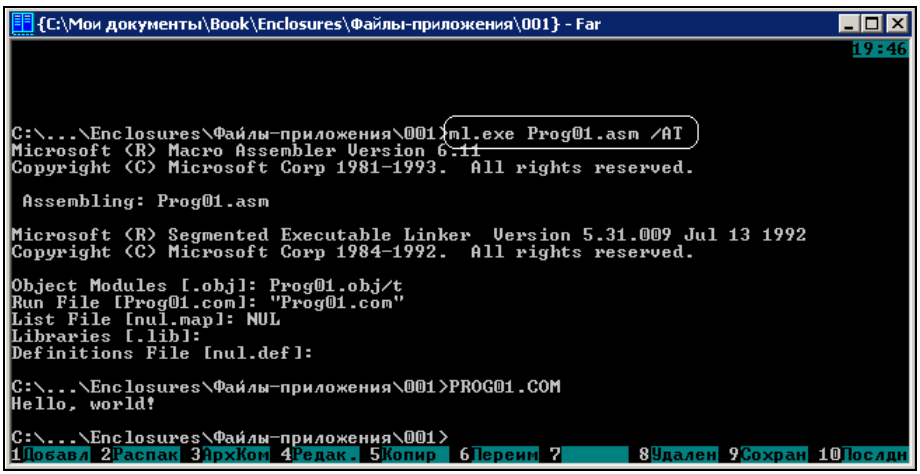


Рис. 1.1. Ассемблирование и результат выполнения программы Prog01.com

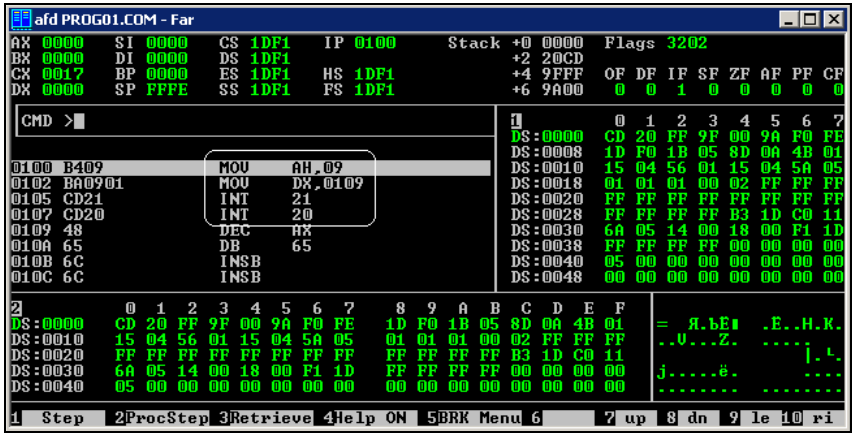


Рис. 1.2. Вид программы в отладчике AFD Pro

На рис. 1.2 показано, как эта программа выглядит в отладчике AFD Pro. Пока не обращайте особого внимания на различие между реальным кодом, набранным руками, и тем, как эта программа отображается в отладчике. Подробно работу отладчика мы рассмотрим в последующих главах.

1.4. Резюме

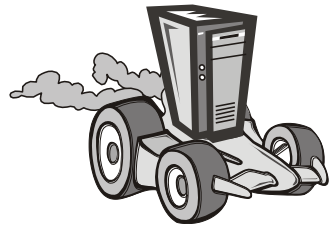
Целью данной главы не было разобраться подробно с каждым оператором. Это невозможно, если вы не обладаете базовыми знаниями. Но, прочитав 3—4 главы, вы поймете принцип и структуру программы на ассемблере.

Может быть, ассемблер вам показался чрезвычайно сложным, но это, поверьте, только с первого взгляда. Вы должны научиться строить алгоритм программы на ассемблере в голове, а для этого нужно будет самостоятельно написать несколько программ, опираясь на информацию из данной книги. Будем постепенно учиться мыслить структурой ассемблера, составлять алгоритмы, программы, используя операторы языка. После изучения очередной главы вы будете чувствовать, что постепенно начинаете осваивать ассемблер, будет становиться все проще и проще.

Например, если вы знакомы с Бейсиком, то, ставя перед собой задачу написать программу, выводящую 10 слов "Привет", вы будете использовать операторы `FOR`, `NEXT`, `PRINT` и пр., которые тут же появятся в ваших мыслях. Вы строите определенный алгоритм программы из этих операторов, который в какой-то степени применим только к Бейсику. То же самое и с ассемблером. При постановке задачи написать ту или иную программу вы мысленно создаете алгоритм, который применим к ассемблеру и только, т. к. языков, похожих на ассемблер, просто не существует. Наша задача — научить вас создавать в уме алгоритмы, применимые к ассемблеру, т. е. образно говоря, научить "мыслить на ассемблере".

* * *

В *главе 2* мы подробно рассмотрим регистры процессора и напишем еще одну простую программу.



Глава 2

Регистры процессора

2.1. Введение в регистры микропроцессоров 8086—80186

Регистр, как мы уже говорили ранее, — это специально отведенная память для временного хранения каких-то данных. Микропроцессоры 8086—80186 имеют 14 регистров. В *главе 1* мы познакомились с двумя из них: `ah` и `dx`. В табл. 2.1, 2.3 и 2.4 приведен перечень всех регистров, кроме `ip` и регистра флагов, которые будут рассмотрены отдельно.

2.1.1. Регистры данных

Регистры данных могут использоваться программистом по своему усмотрению (за исключением некоторых случаев). В них можно хранить любые данные: числа, адреса и пр. В верхнем ряду табл. 2.1 находятся 32-разрядные регистры, которые могут хранить числа от 0 до 4 294 967 295 (0FFFFFFFFh). Их мы будем рассматривать позже. Во втором ряду — 16-разрядные, которые могут хранить числа от 0 до 65 535 или от 0h до FFFFh в шестнадцатеричной системе, что одно и то же.

В следующей строке расположен ряд 8-разрядных регистров: `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh`, `dl`. В эти регистры можно загружать максимальное число 255 (FFh). Это так называемые половинки (старшая или младшая) 16-разрядных регистров.

Таблица 2.1. *Регистры данных*

EAX				**EBX**			
ax		**bx**		**cx**		**dx**	
ah	al	bh	bl	ch	cl	dh	dl
Аккумулятор		База		Счетчик		Регистр данных	

Мы уже изучили оператор `mov`, который предназначен для загрузки числа в регистр. Чтобы присвоить, к примеру, регистру `al` число 35h, нам необходимо записать так:

```
mov al,35h
```

а регистру ax — число 346Ah, так:

```
mov ax,346Ah
```

Если мы попытаемся загрузить большее число, чем может содержать регистр, то при ассемблировании программы произойдет ошибка. Например, следующие записи будут ошибочны:

~~mov ah,123h~~ → максимум FFh

~~mov bx,12345h~~ → максимум FFFFh

~~mov dl,100h~~ → максимум FFh

Здесь надо отметить, что если шестнадцатеричное число начинается не с цифры (12h), а с буквы (C5h), то перед таким числом ставится ноль: 0C5h. Это необходимо для того, чтобы программа-ассемблер могла отличить, где шестнадцатеричное число, а где название переменной или метки. Далее мы рассмотрим это на примере.

Допустим, процессор выполняет команду mov ax,1234h. В этом случае в регистр ah загружается число 12h, а в регистр al — 34h. То есть ah, al, bh, bl, ch, cl, dh и dl — это младшие (**L**ow) или старшие (**H**igh) половинки 16-разрядных регистров (табл. 2.2).

Таблица 2.2. *Результаты выполнения различных команд*

Команда	**Результат**
mov ax,1234h	ax = 1234h, ah = 12h, al = 34h
mov bx,5678h	bx = 5678h, bh = 56h, bl = 78h
mov cx,9ABCh	cx = 9ABCh, ch = 9Ah, cl = 0BCh
mov dx,0DEF0h	dx = 0DEF0h, dh = 0DEh, dl = 0F0h

2.1.2. Регистры-указатели

Регистры si (индекс источника) и di (индекс приемника) используются в строковых операциях. Регистры bp и sp задействуются при работе со стеком (табл. 2.3). Мы подробно их рассмотрим на примерах в следующих главах.

Таблица 2.3. *Регистры-указатели*

si	**Di**	**bp**	**sp**
Индекс источника	Индекс приемника	Регистры для работы со стеком	

2.1.3. Сегментные регистры

Сегментные регистры (табл. 2.4) необходимы для обращения к тому или иному сегменту памяти (например, видеобуферу). Сегментация памяти — довольно сложная и объемная тема, которую также будем рассматривать в следующих главах.

Таблица 2.4. *Сегментные регистры*

CS	**DS**	**ES**	**SS**
Регистр кода	Регистр данных	Дополнительный регистр	Регистр стека

2.2. Команды сложения и вычитания

Для выполнения арифметических операций сложения и вычитания в ассемблере существуют следующие операторы: `add`, `sub`, `inc`, `dec`.

2.2.1. Оператор *add*

Формат оператора `add` показан в табл. 2.5. Впоследствии мы всегда будем оформлять новые команды в подобные таблицы. В столбце **Команда** будет описана новая команда и ее применение. В столбце **Назначение** — что выполняет или для чего служит данная команда, а в столбце **Процессор** — модель (тип) процессора, начиная с которой команда поддерживается. В столбце **Перевод** будет указано, от какого английского слова образовано название оператора, и дан перевод этого слова.

Таблица 2.5. *Оператор* `add`

Команда	Перевод	Назначение	Процессор
`add` *приемник*, *источник*	**Add**ition — сложение	Сложение	8086

В данном примере оператор поддерживается процессором 8086, но работать команда будет, естественно, и на более современных процессорах (80286, 80386, 80486, Pentium и т. д.).

Команда `add` производит сложение двух чисел (листинг 2.1).

Листинг 2.1. Примеры использования оператора `add`

```
mov al,10       ;загружаем в регистр al число 10
add al,15       ;al = 25; al — приемник, 15 — источник
mov ax,25000    ;загружаем в регистр ax число 25000
add ax,10000    ;ax = 35000; ax — приемник, 10000 — источник
mov cx,200      ;загружаем в регистр cx число 200
mov bx,760      ;a в регистр bx — 760
add cx,bx       ;cx = 960, bx = 760 (bx не меняется); cx — приемник,
                ;bx — источник
```

2.2.2. Оператор *sub*

Команда `sub` производит вычитание двух чисел (табл. 2.6, листинг 2.2).

Таблица 2.6. *Оператор* `sub`

Команда	Перевод	Назначение	Процессор
`sub` *приемник*, *источник*	**Sub**traction — вычитание	Вычитание	8086

Листинг 2.2. Примеры использования оператора `sub`

```
mov al,10
sub al,7              ;al = 3; al — приемник, 7 — источник
mov ax,25000
sub ax,10000          ;ax = 15000; ax — приемник, 10000 — источник
mov cx,100
mov bx,15
sub cx,bx
```

ЭТО ИНТЕРЕСНО

Следует отметить, что ассемблер — максимально быстрый язык. Можно посчитать, сколько раз за одну секунду процессор сможет сложить два любых числа от 0 до 65 535.

Каждая команда процессора выполняется определенное количество тактов. Когда говорят, что тактовая частота процессора 100 МГц, то это значит, что за секунду проходит 100 миллионов тактов. Чтобы компьютер сложил два числа, ему нужно выполнить следующие команды:

```
...
mov ax,2700
mov bx,15000
add ax,bx
...
```

В результате выполнения данных инструкций в регистре `ax` будет число 17 700, а в регистре `bx` — 15 000. Команда `add ax,bx` выполняется за один такт на процессоре 80486. Получается, что компьютер 486 DX2-66 МГц за одну секунду сложит два любых числа от 0 до 0FFFFh 66 миллионов (!) раз!

2.2.3. Оператор *inc*

Формат оператора `inc` представлен в табл. 2.7.

Таблица 2.7. *Оператор* `inc`

Команда	Перевод	Назначение	Процессор
`inc` *приемник*	**Inc**rement — инкремент	Увеличение на единицу	8086

Команда `inc` увеличивает на единицу содержимое приемника (регистра или ячейки памяти). Она эквивалентна команде:

```
add источник, 1
```

только выполняется быстрее на старых компьютерах (до 80486) и занимает меньше байтов (листинг 2.3).

Листинг 2.3. Примеры использования оператора `inc`

```
mov al,15
inc al          ;теперь al = 16 (эквивалентна add al,1)
mov dh,39h
inc dh          ;dh = 3Ah (эквивалентна add dh,1)
mov cl,4Fh
inc cl          ;cl = 50h (эквивалентна add cl,1)
```

2.2.4. Оператор *dec*

Формат оператора `dec` представлен в табл. 2.8.

Таблица 2.8. *Оператор* `dec`

Команда	Перевод	Назначение	Процессор
`dec` *приемник*	**Dec**rement — декремент	Уменьшение на единицу	8086

Команда `dec` уменьшает на единицу содержимое приемника (листинг 2.4). Она эквивалентна команде:

`sub` *источник*`, 1`

Листинг 2.4. Примеры использования оператора `dec`

```
mov al,15
dec al           ;теперь al = 14
mov dh,3Ah
dec dh           ;dh = 39h
mov cl,50h
dec cl           ;cl = 4Fh
```

2.3. Программа для практики

Рассмотрим одну небольшую программу, которая выводит на экран сообщение и ждет, когда пользователь нажмет любую клавишу. После чего возвращается в DOS.

Работать с клавиатурой позволяет прерывание BIOS (ПЗУ) `16h`, которое можно вызывать даже до загрузки операционной системы, в то время как прерывания `20h`, `21h` и пр. доступны только после загрузки IO.SYS/MSDOS.SYS — определенной части ОС MS-DOS.

Чтобы заставить программу ждать нажатия пользователем любой клавиши, следует вызвать функцию `10h` прерывания `16h`. Вот как это выглядит на практике:

```
mov ah,10h  ;в ah всегда указывается номер функции
int 16h     ;вызываем прерывание 16h — сервис работы с клавиатурой BIOS (ПЗУ)
```

После нажатия любой клавиши компьютер продолжит выполнять программу, а регистр `ax` будет содержать код клавиши, которую нажал пользователь.

Следующая программа (\002\prog02.asm) выводит на экран сообщение и ждет нажатия любой клавиши, что равнозначно команде `PAUSE` в BAT-файлах (листинг 2.5).

Листинг 2.5. Программа для практики

```
(01) CSEG segment
(02) org 100h
(03) Start:
(04)
(05)        mov ah,9
(06)        mov dx,offset String
(07)        int 21h
(08)
(09)        mov ah,10h
(10)        int 16h
(11)
(12)        int 20h
(13)
(14) String db 'Нажмите любую клавишу...$'
(15) CSEG ends
(16) end Start
```

Строки с номерами (01), (02) и (15) пока опускаем. В строках (05)—(07), как вы уже знаете, производится вывод строки на экран. Затем (строки (09), (10)) программа ждет нажатия клавиши. И наконец, строка (12) завершает работу нашей программы.

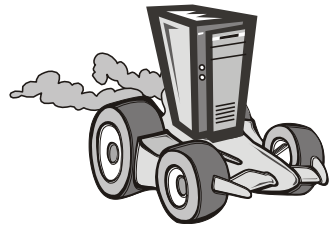
Мы уже изучили операторы `inc`, `dec`, `add` и `sub`. Вы можете поэкспериментировать (лучше в отладчике) с числами. Например, вот так:

```
...
mov ah,0Fh
inc ah
int 16h
...
```

Это позволит вам лучше запомнить новые операторы.

* * *

В *главе 3* рассмотрим двоичную систему счисления, основы сегментации памяти и сегментные регистры. Напишем интересную программу.



Глава 3

Сегментация памяти в реальном режиме

В данной главе мы рассмотрим основополагающие принципы программирования на языке ассемблера. Необходимо тщательно разобраться в каждом предложении, уяснить двоичную систему счисления и понять принцип сегментации памяти в реальном режиме. Мы также рассмотрим операторы ассемблера, которые не затрагивали в примерах из предыдущих глав. Сразу отмечу, что это одна из самых сложных глав данной книги. Автор попытался объяснить все как можно проще, избегая сложных определений и терминов. Если что-то не поняли — не пугайтесь! Со временем все станет на свои места. Если вы полностью разберетесь с материалом данной главы, то считайте, что базу ассемблера вы изучили. Начиная с *главы 4*, будем изучать язык намного интенсивней.

Для того чтобы лучше понять сегментацию памяти, нам нужно воспользоваться отладчиком. Лучше использовать в работе два отладчика: CodeView (CV.EXE) и AFD Pro (AFD.EXE). Допустим, вы написали программу на ассемблере и назвали ее prog03.asm. Сассемблировав, вы получили файл prog03.com. Тогда, чтобы запустить программу под отладчиком CodeView/AFD, необходимо набрать в командной строке MS-DOS следующее:

```
CV.EXE prog03.com
```

либо:

```
AFD.EXE prog03.com
```

Итак, вдохните глубже и — вперед!

3.1. Двоичная система счисления. Бит и байт

Рассмотрим, как в памяти компьютера хранятся данные. Вообще, как компьютер может хранить, например, слово "диск"? Главный принцип — намагничивание и размагничивание одной дорожки (назовем это так). Одна микросхема памяти — это, грубо говоря, огромное количество дорожек (примерно как на магнитофонной кассете). Сейчас попробуем разобраться.

Предположим, что:

```
Ноль будет обозначаться как 0000 (четыре нуля),
Один                        0001,
Два                         0010 (т. е. правую единицу меняем на ноль,
а вторую устанавливаем в 1).
```

Далее так:

```
Три             0011
Четыре          0100
Пять            0101
Шесть           0110
Семь            0111
Восемь          1000
Девять          1001
```

и т. д.

"Нули" и "единицы" — это так называемые биты. Один бит, как вы уже заметили, может иметь только два значения — 0 или 1, т. е. размагничена или намагничена та или иная дорожка ("0" и "1" — это условное обозначение). Если внимательно посмотреть, то можно обнаружить, что каждый следующий установленный бит, начиная справа, увеличивает число в два раза: 0001 в нашем примере — один; 0010 — два; 0100 — четыре; 1000 — восемь и т. д. Это и есть двоичная форма представления данных. Чтобы обозначить числа от 0 до 9, нам нужно четыре бита (хоть они и не будут до конца использованы; можно было бы продолжить: десять — 1010, одиннадцать — 1011, ..., пятнадцать — 1111).

Компьютер хранит данные в памяти именно так. Для обозначения какого-нибудь символа (цифры, буквы, запятой, точки и др.) компьютер использует определенное количество бит. Компьютер "распознает" 256 (от 0 до 255) различных символов по их коду. Этого достаточно, чтобы вместить все цифры (0—9), буквы латинского алфавита (a—z, A—Z), русского (а—я, А—Я) и др. *(см. приложение 3)*. Для представления символа с максимально возможным кодом (255) нужно 8 бит. Эти 8 бит называются байтом. Таким образом, один любой символ — это всегда 1 байт (табл. 3.1).

Таблица 3.1. *Один байт с кодом символа "Z"*

0	1	0	1	1	0	1	0
P	H	P	H	H	P	H	P

ПРИМЕЧАНИЕ

Символы "H" и "P" в таблице обозначают "намагничено" или "размагничено" соответственно.

Можно элементарно проверить. Создайте в текстовом редакторе файл с любым именем и напечатайте в нем один символ, например, "M", но не нажимайте клавишу <Enter>. Если вы посмотрите его размер, то файл будет равен 1 байту. Если ваш редактор позволяет смотреть файлы в шестнадцатеричном формате, то вы сможете узнать и код сохраненного символа. В данном случае — большая буква "M" имеет код `4Dh` в шестнадцатеричной системе, которую мы уже знаем, или 1001101 в двоичной. Таким образом, слово "диск" будет занимать 4 байта или $4 \times 8 = 32$ бита. Как вы уже поняли, компьютер хранит в памяти не сами буквы (символы) этого слова, а последовательность "единичек" и "ноликов".

"В таком случае, почему на экране мы видим набор символов (текст, предложения, слова), а не "единички-нолики"?" — спросите вы. Чтобы удовлетворить ваше любопытство, забежим немного вперед и отметим, что всю работу по выводу самого символа (а не битов) на экран выполняет видеокарта (видеоадаптер), которая находится в вашем компьютере. И если бы ее не было, то мы, естественно, на экране ничего бы не увидели.

В ассемблере после двоичного числа всегда должен стоять символ b. Это нужно для того, чтобы в процессе обработки нашего файла ассемблер-программа смогла различать десятичные, шестнадцатеричные и двоичные числа. Например: 10 — это десять, 10h — это шестнадцать, а 10b — это два. Таким образом, в регистры можно загружать двоичные, десятичные и шестнадцатеричные числа. Например:

```
...
mov ax,20
mov bh,10100b
mov cl,14h
...
```

В результате в регистрах `ax`, `bh` и `cl` будет находиться одно и то же число, только загружаем мы его, используя разные системы счисления. В компьютере же оно будет храниться в двоичном формате (как в регистре `bh`).

Итак, подведем итог. В компьютере вся информация хранится в двоичном формате (двоичной системе) примерно в таком виде: 10101110 10010010 01111010 11100101 (естественно, без пробелов; для наглядности мы разделили байты). *Восемь бит — это один байт*. Один символ занимает один байт, т. е. восемь бит. По идее, ничего сложного нет. Очень важно уяснить данную тему, т. к. мы будем постоянно пользоваться двоичной системой, и вам необходимо знать ее на "отлично". В принципе, даже если что-то не совсем понятно, то — не отчаивайтесь! Со временем все станет на свои места.

3.1.1. Как перевести двоичное число в десятичное

Чтобы перевести двоичное число в десятичное, надо сложить двойки в степенях, показатели которых соответствуют позициям единиц в двоичном числе.

Например, возьмем число 20. В двоичной системе оно имеет следующий вид: 10100b.

Итак, начнем слева направо, считая от 4 до 0. Число в нулевой степени всегда равно единице:

$$10100b = 2^4 + 0 + 2^2 + 0 + 0 = 16 + 8 = 20$$

либо

$$10100b = 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1 = 16 + 0 + 4 + 0 + 0 = 20.$$

3.1.2. Как перевести десятичное число в двоичное

Можно делить его на два, записывая остаток справа налево:

20/2 = 10, остаток 0;

10/2 = 5, остаток 0;