Шибалкин Д. В.

Курс «Микропроцессоры».

Практические занятия по программированию на ассемблере.

Калининград 2007 г.

Введение. Двоичная и шестнадцатеричная системы счисления.

Двоичная система. Системой счисления называют символический метод записи чисел с помощью письменных знаков. В привычной для нас десятичной системе числа записываются при помощи десяти различных знаков – цифрами от «0» до «9»:

```
12, 345, 65532
```

Положение знака принято называть разрядом. В десятичной системе разряды имеют название. Например, для числа 65532:

Десятки тысяч	Тысячи	Сотни	Десятки	Единицы
6	5	5	3	6

Кроме того, разряд расположенный левее, считается «старшим» по отношению к предыдущему.

Наряду с десятичной существуют и другие системы счисления, в которых используется различное число знаков. Одна из них – двоичная система – использует всего два знака «0» и «1». Рассмотрим правила перевода двоичного числа в десятичное:

```
110101011
```

Пронумеруем каждый разряд двоичного числа от нуля справа налево:

Разряд	8	7	6	5	4	3	2	1	0
	1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**

Затем каждому разряду припишем «вес», равный 2^N, где N – номер разряда:

Вес	256	128	64	32	16	8	4	2	1
Разряд	8	7	6	5	4	3	2	1	0
	1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**

Теперь для перевода в десятичное число необходимо вычислить сумму произведений значений разрядов на их веса, т.е.:

```
1*256 + 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 0*4 + 1*2 + 1*1 = 427
```

Т.о. двоичному числу «110101011» соответствует десятичное число «427». Отметим, что с помощью N разрядов в двоичной системе счисления можно записать 2^N различных чисел (т.о. различных комбинаций «0» и «1») от 0 до $(2^N – 1)$:

```
8 разрядов:  0 – 255 (256 значений)
16 разрядов: 0 – 65535 (65536 значений)
20 разрядов: 0 – 1048575 (1048576 значений)
32 разряда:  0 – 4294967295 (4294967296 значений)
```

Для перевода десятичного числа в двоичное необходимо осуществить последовательное деление на 2, выделяя целую часть результата и остаток. Рассмотрим пример для числа «374»:

```
374/2 = 187, остаток 0
187/2 = 93,  остаток 1
93/2  = 46,  остаток 1
46/2  = 23,  остаток 0
23/2  = 11,  остаток 1
11/2  = 5,   остаток 1
5/2   = 2,   остаток 1
2/2   = 1,   остаток 0
1/1   = 0,   остаток 1
```

Значения остатков соответствуют значениям разрядов двоичного числа. При этом остаток, полученный первым, соответствует младшему разряду числа, а остаток полученный последним – старшему. Т.о. десятичному числу «374» соответствует двоичное число «101110110».

Двоичная система используется в цифровых устройствах, для хранения и передачи данных, поскольку является одной из самых простых. Чаще всего информация записывается и передаётся не непрерывным потоком «0» и «1», а порциями по восемь знаков, или «бит». Совокупность восьми бит называют «байтом», и именно в байтах чаще всего измеряют объём информации. Один байт т.о. может принимать значения от 0 до 255. Если требуется записать или передать большее число, то используется несколько байт. Например, число «54237» в двоичной системе счисления записывается как:

```
54237 - 1101001111011101
```

и его можно представить как комбинацию двух байт, которую называют «словом»:

```
1101001111011101 – 11010011 11011101
     54237           211      221
```

Байт «211» в данном случае называют старшим байтом слова, а «221» – младшим. Кроме байта используют также кратные байту единицы объёма информации:

```
1 килобайт (Кб) = 1024 байт
1 мегабайт (Мб) = 1024 Кб
1 гигабайт (Гб) = 1024 Мб
1 терабайт (Тб) = 1024 Гб
```

Шестнадцатеричная система. В программировании и в компьютерной документации чаще всего используется другая система счисления – шестнадцатеричная. В этой системе для записи числа используется 16 знаков: цифры от «0» до «9» и буквы латинского алфавита от «A» до «F». Соответствие знаков шестнадцатеричной системы и чисел десятичной системы счисления следующее:

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Правила перевода шестнадцатеричного числа в десятичное аналогичны правилам перевода двоичного числа:

```
2A46BF
```

Снова пронумеруем каждый разряд числа от нуля справа налево:

Разряд	5	4	3	2	1	0
	2	**A**	**4**	**6**	**B**	**F**

Каждому разряду припишем «вес», равный 16^N, где N – номер разряда:

Вес	1048576	65536	4096	256	16	1
Разряд	5	4	3	2	1	0
	2	**A**	**4**	**6**	**B**	**F**

Теперь вычислим сумму произведений значений разрядов на их веса, т.е.:

```
2*1048576 + 10*65536 + 4*4096 + 6*256 + 11*16 + 1*15 = 2770623
```

Т.о. шестнадцатеричному числу «2A46BF» соответствует десятичное число «2770623». Перевод из десятичной системы в шестнадцатеричную также аналогичен переводу десятичного числа в двоичное и осуществляется последовательным делением на 16. Например, для числа «194731»:

```
194731/16 = 12170, остаток 11 (B)
12170/16  = 760,   остаток 10 (A)
760/16  = 47,      остаток 8  (8)
47/16  = 2,        остаток 15 (F)
2/16  = 0,         остаток 2  (2)
```

Снова остаток, полученный первым, будет соответствовать младшему разряду числа, а остаток полученный последним – старшему. Т.о. десятичному числу «194731» соответствует шестнадцатеричное число «2F8AB». Для того чтобы различать десятичные и шестнадцатеричные числа при написании, к шестнадцатеричным числам прибавляется либо префикс «0x», либо окончание «h». Например: «0x2F8AB» или «2F8ABh».

Как уже было отмечено, единицей объёма информации является байт. Пусть нам необходимо сохранить или передать число «34541». Переведём в двоичную систему счисления:

```
34541 – 1000011011101101 – 10000110 11101101
                             134      237
```

Видно, что десятичная форма записи не отражает, как это число будет представлено в виде двух байт. Теперь переведём «34541», «134» и «237» в шестнадцатеричную систему счисления:

```
34541 – 86ED
134   - 86
237   - ED

34541 – 86ED – 86 ED
```

Видно, что каждые два символа шестнадцатеричного числа соответствуют одному байту. Т.е. запись числа в шестнадцатеричной форме показывает, как это число будет представлено в виде отдельных байт. Нетрудно также заметить, что каждый шестнадцатеричный символ соответствует четырём битам двоичной системы:

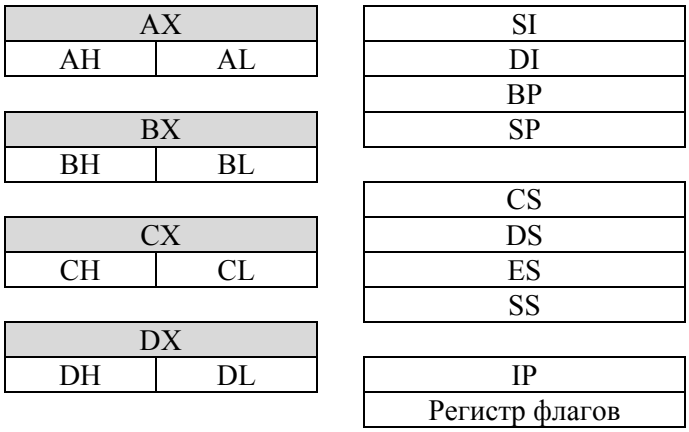
```
34541 – 86ED – 1000011011101101 – 1000 0110 1110 1101
                                   8    6    14   13
                                   8    6    E    D
```

Именно два последних обстоятельства и обусловили широкое распространение шестнадцатеричной системы счисления.

I. Общие сведения о реальном режиме работы микропроцессоров семейства x86.

1.1. Регистры. В реальном режиме микропроцессоры семейства x86 работают аналогично процессору 8086. Как известно, этот процессор характеризуется наличием 16-разрядной шины данных и 20-разрядной адресной шины. Это означает, что для задания адреса конкретной ячейки памяти используется 20 бит, и, следовательно, максимальный объём адресуемой памяти составляет 2^{20} = 1048576 байт, или 1 Мб. Каждый внутренний регистр процессора может хранить 16 бит информации, т.е. два байта или одно слово.

Рассмотрим регистры микропроцессора 8086:



Регистры AX, BX, CX, DX называют регистрами общего назначения. AX называют аккумулятором, BX – базовым регистром, CX – счётчиком, DX – регистром данных. Каждый из этих регистров условно разделяют на два 8-разрядных – старшую и младшую часть. Например, AL – младшая, AH – старшая часть регистра AX. По мере необходимости младшие и старшие части регистров общего назначения могут использоваться по отдельности.

Регистры CS, DS, ES, SS называют сегментными регистрами, BP – указатель базы, SP – указатель стека, SI – индекс источника, DI – индекс приёмника, IP – указатель команд.

Особое значение имеет регистр флагов. В отличие от других регистров, отдельные биты регистра флагов служат для обозначения некоторого состояния процессора. Перечислим три наиболее важных для нас флага:

- 0-й бит регистра флагов (младший) называют «флагом переноса» или CF. Он установлен (т.е. равен 1), если выполнение некоторой операции завершилось с ошибкой, и сброшен (т.е. равен 0) в противном случае
- 6-й бит называют «флагом нуля» или ZF. Он установлен, если результат последней операции (напр. вычитания) равен нулю, и сброшен в противном случае
- 7-й бит носит название «флага знака» или SF. Он устанавливается в том случае, кода результат последней операции оказывается отрицательным, в противном случае он сброшен

Подробнее назначение каждого из регистров мы рассмотрим позже на конкретных примерах.

1.2. Адресация памяти. В силу наличия 16-разрядных регистров микропроцессор не может хранить в них 20-разрядные адреса для операций с конкретной ячейкой памяти. Для хранения адреса используется два 16-разрядных числа: сегментный адрес и смещение. Сегментный адрес (или просто сегмент) – это кратный 16-ти адрес некоторой ячейки памяти, уменьшенный в 16 раз. Например, для ячейки памяти с адресом 0x1BE20 сегментный адрес будет 0x1BE2. Смещение – это 16-разрядное число, обозначающее смещение относительно сегментного адреса. Т.о. пара 16-разрядных чисел – сегмент и смещение – могут задать адрес любой ячейки памяти. Например, ячейка памяти с адресом 0x1BE2C может быть задана как:

`0x1BE2` – сегментный адрес
`0x000C` – смещение относительно сегментного адреса

Обычно адрес записывают в форме «сегмент:смещение». Т.е. адрес 0x1BE2C запишется как:

```
1BE2:000Ch
```

Как следует из определений, сегмент и смещение – 16-разрядные числа, и они могут быть сохранены в регистрах микропроцессора. Чтобы из сегментного адреса и смещения восстановить полный адрес в памяти необходимо сегментный адрес умножить на 16 (в шестнадцатеричной системе счисления для этого достаточно приписать справа 0, т.к. 16 = 0x10) и прибавить к полученному числу смещение. Для нашего примера:

```
1BE2:000Ch
1BE2h*10h = 1BE20h
1BE20h + 000Ch = 1BE2Ch
```
или `0x1BE2C`

Отметим, что такое задание адреса неоднозначно – та же ячейка памяти 0x1BE2C может быть задана как:

`1BE1:001Ch, 1BE0:002Ch, 1BDF:003Ch,` и т.д.

1.3. Выполнение программ. Для того чтобы процессор выполнил определённую программу, она должна находиться в оперативной памяти. Программа состоит из последовательностей команд, каждая из которых в свою очередь состоит из кода операции и операнда. Код операции – это число, обозначающее действие, которое должен выполнить процессор. Операнд – это данные, над которыми выполняется операция. Записанная таким образом программа называется машинным кодом. Приведём пример программы, которая в регистр AX заносит слово 0x0010, в регистр BH – байт 0x12, в регистр BL – байт 0xFF, а затем складывает содержимое AX и BX:

```
B8 10 00 B7 12 B3 FF 03 C3
```

Здесь:

`B8` код операции «записать в AX слово …»
`10 00` слово 0x0010[1]

`B7` код операции «записать в BH байт …»
`12` байт 0x12

`B3` код операции «записать в BL байт …»
`FF` байт 0xFF

`03 C3` код операции «сложить AX с регистром BX»

Т.о. эта программа состоит из 4-х команд. Пусть эта программа загружена в память, начиная с адреса 0x01BE0:

Адрес/смещение	0	1	2	3	4	5	6	7	8
0x01BE0	B8	10	00	B7	12	B3	FF	03	C3

Для выполнения программы процессору необходимо сообщить этот адрес (или «точку входа»). Для этого в регистр CS заносится сегмент, а регистр IP – смещение соответствующие этому адре-

[1] В памяти всегда сначала идёт младший байт слова, а затем старший.

су. В данном случае это 01BE:0000h. Процессор на основе содержимого этих регистров вычисляет полный адрес, по которому записан код операции. В данном случае это 0x01BE0. Операция B8 «записать в AX слово …» предполагает, что за ней в качестве операнда располагается слово данных. Т.о. первая команда занимает в памяти 3 байта, и процессор считывает их, начиная с адреса 0x01BE0. После выполнения операции содержимое IP увеличивается на 3 и становится равным 0x0003 (значение CS не меняется, оно по-прежнему равно 0x01BE). Т.о. процессор определяет, что код следующей операции располагается по адресу 01BE:0003h, т.е. 0x01BE3. Операция B7 «записать в BH байт …» предполагает, что за ней в качестве операнда располагается байт данных. Следовательно, вторая команда занимает в памяти 2 байта, и процессор считывает их, начиная с адреса 0x01BE3. После выполнения операции содержимое IP увеличивается на 2 и становится равным 0x0005. Далее снова происходит считывание команды из памяти, её выполнение, увеличение IP на соответствующую величину, и т. д. Последние две команды располагаются по адресам 01BE:0005h (0x01BE5) и 01BE:0007h (0x01BE7) и, как видно, они тоже 2-хбайтные. После выполнения 3-х первых команд в AX будет находиться слово 0x0010, а в BX – слово 0x12FF (BH – 0x12, BL – 0xFF). При выполнении последней команды процессор сложит эти числа и сохранит результат в AX (это будет слово 0x130F).

Из приведённого выше примера следует, что сегментный регистр CS служит для хранения сегментного адреса программы, а IP – смещение, относительно CS, указывающее на команду, которую необходимо выполнить. После выполнения очередной команды значение IP увеличивается на число байт, равному размеру команды. По этой причине CS называют «сегментным регистром кода», а IP – «указателем команд».

Как было отмечено выше, для того чтобы процессор выполнил программу, её нужно загрузить в оперативную память и сообщить процессору начальный адрес программы (точку входа). Все эти действия выполняет операционная система.

1.4. BIOS и MS-DOS. Распределение оперативной памяти в MS-DOS. После включения питания компьютера процессор сначала выполняет программу, записанную в ROM BIOS (ROM – Read Only Memory, «память только для чтения», BIOS – Basic Input Output System, «базовая система ввода вывода»). Эта программа осуществляет обнаружение и тестирование основных устройств компьютера. Информация об их состоянии сохраняется в оперативной памяти в диапазоне 0x00400 – 0x00500 (эту область называют «область данных BIOS»). После этого осуществляется копирование самого BIOS в оперативную память в области 0xC0000 – 0xD0000 и 0xE0000 – 0xFFFFF. В состав BIOS входят готовые программы, выполняющие обслуживание и управление основных устройств компьютера. Начальные адреса (точки входа) этих программ записываются в область оперативной памяти 0x00000 – 0x00400 (т.е. первые 1024 байта) по 4 байта на адрес. Эта область называется «таблицей векторов прерываний», записанные в ней адреса – «векторами прерываний», а сами программы «прерываниями» или «обработчиками прерываний». В дальнейшем прерывания используются операционной системой и пользовательскими программами. Отметим, что векторы прерываний BIOS занимают только часть таблицы.

После загрузки в оперативную память BIOS начинается загрузка операционной системы с диска. Рассмотрим загрузку MS-DOS (Microsoft DOS, DOS – Disk Operating System, «дисковая операционная система»). Ядро операционной системы MS-DOS составляют файлы IO.SYS, MSDOS.SYS и COMMAND.COM. Часть содержимого этих файлов помещается в память, начиная с адреса 0x00700. При этом часть файла COMMAND.COM загружается в область памяти размером около 5 Кб с верхней границей 0x09FFFF. Это так называемая «транзитная часть» COMMAND.COM. Она ответственна за обработку команд водимых с клавиатуры пользователем. Помимо самой MS-DOS в память могут загружаться дополнительные программы поддержки работы мыши, CD-ROM, и др. (т.н. «драйверы»). В среднем объём памяти занимаемый MS-DOS обычно равен нескольким десяткам килобайт. При загрузке MS-DOS дополняет таблицу векторов прерываний своими прерываниями. Сами прерывания располагаются в той же области памяти, что и сама MS-DOS, начиная с адреса 0x00700. В отличие от BIOS-прерываний, DOS-прерывания предназначены в основном для работы с файлами, манипулирования памятью, вывода текстовой информации на экран и ввода с клавиатуры. Также MS-DOS заполняет область 0x00500 – 0x00700 («область данных DOS»), в которой аналогично «области данных BIOS» хранится информация о состоянии системы.

Часть оперативной памяти отводится под видеобуферы (текстовый и графический) в которых хранится образ того, что отображается в данный момент на экране. Любые операции вывода на экран (рисование графических изображений или вывод текста) осуществляется путём изменения содержимого соответствующего видеобуфера. Графический видеобуфер располагается в области 0xA0000 – 0xB0000, текстовый в области 0xB8000 – 0xC0000.

Т.о. распределение памяти в MS-DOS можно представить следующим образом:

Начало	Конец	Размер	Содержимое	
0x00000	0x003FF	1024 байта	Таблица векторов прерываний	Стандартная память 640 Кб
0x00400	0x004FF	256 байт	Область данных BIOS	
0x00500	0x006FF	512 байт	Область данных DOS	
0x00700	–	около 20 Кб	IO.SYS, MS-DOS.SYS	
–	–	?	Драйверы	
–	–	около 4 Кб	COMMAND.COM	
–	–	?	Свободно	
–	0x09FFF	около 5 Кб	COMMAND.COM (транзитная часть)	Верхняя память (UMB) 384 Кб
0xA0000	0xAFFFF	64 Кб	Графический видеобуфер	
0xB0000	0xB7FFF	32 Кб	Свободно	
0xB8000	0xBFFFF	32 Кб	Текстовый видеобуфер	
0xC0000	0xCFFFF	64 Кб	BIOS	
0xD0000	0xDFFFF	64 Кб	Свободно	
0xE0000	0xFFFFF	128 Кб	BIOS	
0x100000	0x10FFFF	64 Кб	Свободно	Старшая память (HMA)
0x110000	–	около 1 Мб	Свободно	Расширенная память (XMS)
–	–	до 32 Мб	Свободно	Дополнительная память (EMS)

Отметим, что для пользовательских программ непосредственно доступна только свободная область в стандартной памяти. В ранних версиях MS-DOS (до версии 5.0) вся верхняя память предназначалась исключительно для BIOS и видеобуферов. В более поздних версиях MS-DOS свободные блоки верхней памяти UMB (Upper Memory Blocks) могут использоваться драйверами вместо стандартной памяти. Для этого в состав MS-DOS включён драйвер памяти HIMEM.SYS и EMM386.EXE (сами они всегда загружается в стандартную память). Перемещение драйверов в верхнюю память увеличивает объём доступной стандартной памяти для пользовательских программ.

Первые 64 Кб за пределами 1 Мб (0x100000 – 0x10FFFF) носят название области старшей памяти HMA (High Memory Area). Несмотря на то, что она находится за пределами мегабайтного адресного пространства, она может быть адресована в реальном режиме. Если определить сегмент 0xFFFF, то область со смещениями 0x0000 – 0x000F соответствует последним 16 байтам верхней памяти:

```
FFFF:0000h  =  FFFFh*10h + 0000h = FFFF0h + 0000h = FFFF0h
FFFF:000Fh  =  FFFFh*10h + 000Fh = FFFF0h + 000Fh = FFFFFh
```

т.е. диапазон 0xFFFF0 – 0xFFFFF.

Область же со смещениями 0x0010 – 0xFFFF находится за пределами 1 Мб:

```
FFFF:0010h  =  FFFFh*10h + 0010h = FFFF0h + 0010h = 100000h
FFFF:FFFFh  =  FFFFh*10h + FFFFh = FFFF0h + 000Fh = 10FFFFh
```

т.е. диапазон 0x100000 – 0x10FFFF.

Начиная с версии 5.0 в MS-DOS благодаря драйверу HIMEM.SYS появилась возможность использования области HMA для хранения части ядра операционной системы, что увеличивает объём свободной стандартной памяти.

За HMA располагается расширенная память XMS (Extended Memory) и дополнительная или отображаемая память EMS (Expanded Memory). Расширенная память может быть доступна только в защищённом режиме работы процессора. Дополнительная память может быть доступна в реальном режиме при использовании драйвера EMM386.EXE и процессоров 80386 и старше. При этом происходит её «отображение» на блоки UMB, т.е. программы используют адреса верхней памяти (в пределах 1 Мб), а EMM386 переносит данные в дополнительную память. Т.о. дополнительная память не может быть вся доступна в реальном режиме одновременно. Объём одновременно доступной дополнительной памяти зависит от объёма свободной верхней памяти. По мере необходимости EMM386 отображает на UBM то одни, то другие участки дополнительной памяти.

II. Основы программирования на языке ассемблер в реальном режиме под MS-DOS.

2.1. Программы типа COM и EXE. Машинный код большинства программ хранится в т.н. «исполняемых» файлах с расширениями COM и EXE. Перед выполнением программы операционная система извлекает из исполняемого файла машинный код, помещает его в оперативную память и устанавливает соответствующие значения регистров CS и IP (задаёт точку входа).

Программы COM и EXE принципиально различаются между собой по принципу использования оперативной памяти. Рассмотрим сначала загрузку программы COM. Пусть машинный код программы занимает 128 байт, и операционная система поместила программу в область памяти с начальным адресом 0x0B150. Вот как будет распределена эта область:

Начало	Конец	Размер	Содержимое
0B15:0000h	0B15:00FFh	256 байт	PSP
0B15:0100h	0B15:017Fh	128 байт	Машинный код программы
0B15:0180h	0B15:FFFDh	65 150 байт	Стек
0B15:FFFEh	0B15:FFFFh	2 байта	0

Сегментные регистры и регистры-указатели будут иметь следующие начальные значения:

CS = 0x0B15, IP = 0x0100, SS = 0x0B15, SP = 0xFFFE, DS = ES = 0x0B15

Здесь все адреса записаны в форме «сегмент:смещение». Первые 256 байт отводятся под «префикс программного сегмента» PSP – эта область необходима операционной системе. Далее следует сама программа. Оставшееся место до смещения 0xFFFD отводится по стек – область памяти для временного хранения данных. Как видно, точка входа располагается по смещению 0x0100, поэтому в регистр CS будет записано 0x0B15 (сегментный адрес), а регистр IP – 0x0100. Использование стека осуществляется следующим образом: перед началом выполнения программы в регистр SS заносится сегментный адрес 0x0B15, а в SP – всегда 0xFFFE. При получении команды, например, «сохранить указанное слово в стеке» процессор уменьшит значение SP на 2 (оно станет 0xFFFC), а затем запишет слово данных по адресу SS:SP (т.е. 0B15:FFFCh). Если возникнет необходимость сохранить в стеке ещё слово, то процессор снова уменьшит значение SP на 2 (оно станет 0xFFFA) и запишет данные по адресу SS:SP (т.е. 0B15:FFFAh), и т.д. При получении же команды «извлечь слово из стека» будет считано слово по адресу SS:SP, а затем процессор увеличит SP на 2. Такой способ временного хранения данных оптимален с точки зрения быстродействия – чтобы сохранить

или извлечь данные из стека нужно только указать операцию (сохранение/извлечение) и сами данные. Адрес же, куда данные будут записываться или откуда извлекаться, уже сохранён в регистрах SS (сегмент) и SP (смещение). При использовании стека необходимо соблюдать только одно условие – данные должны извлекаться в порядке обратном тому, в котором они туда заносились. Другой особенностью стека является то, что он заполняется с конца выделенной ему области памяти. Регистр SP т.о. указывает на верхнюю границу стека и называется «указателем стека», а регистр SS – «сегментным регистром стека». Отметим также, что у процессора 8086 данные в стеке сохраняются только по словам.

Программа типа COM использует всегда только один сегмент – в нём размещается и PSP, и стек, и сама программа, причём размер стека всегда равен:

```
Размер стека = 64 Кб – 258 байт – (размер программы)
```

Т.е. независимо от фактического размера программы, в памяти она всегда занимает 64 Кб. В том же сегменте размещаются и данные (напр. текст, который программа будет выводить на экран).

Программа типа EXE отличается от COM тем, что для PSP, данных и стека выделяются отдельные сегменты. При этом размер стека задаётся программно. Если, например, в программе EXE, фактический размер программы равен 128 байт, размер используемых данных 64 байта, а под стек отведено 512 байт, то после загрузки её по адресу 0x0B150, память будет распределена так:

Начало	Конец	Размер	Содержимое
0B15:0000h	0B15:00FFh	256 байт	PSP
0B25:0000h	0B25:003Fh	64 байт	Данные
0B29:0000h	0B29:01FFh	512 байт	Стек
0B49:0000h	0B49:007Fh	128 байт	Машинный код

Как видно, сегментные адреса областей PSP, данных, кода и стека имеют различные значения. При этом сегментные регистры и регистры-указатели будут иметь следующие начальные значения:

CS = 0x0B55, IP = 0x0000, SS = 0x0B35, SP = 0x0200, DS = ES = 0x0B15

Для хранения сегмента данных используется отдельный регистр DS – «сегментный регистр данных». Смещение в сегменте данных обычно хранится в регистре SI или DX. Т.о. когда необходимо обратится к данным, используется пара регистров DS:SI или DS:DX. Программа типа EXE может иметь как несколько сегментов кода, так и несколько сегментов данных. Для выполнения фрагментов программы, содержащихся в различных сегментах, предусмотрена возможность программного изменения регистров CS и IP специальными командами. Регистр DS также может программно модифицироваться, что позволяет использовать данные, находящиеся в различных сегментах данных. Кроме, того для этой цели может использоваться регистр ES («регистр дополнительного сегмента данных»). Т.е. если сохранить в DS один сегмент данных, а в ES – другой, то появляется возможность использовать данные сразу обоих этих сегментов, без последующей модификации DS и ES. Регистр ES обычно используется в паре с регистрами DI и DX как ES:DI и ES:DX (т.е. регистр DI также служит для хранения смещения). Заметим, что если предполагается использовать регистры DS и ES, то их значения необходимо устанавливать программно, т.к. по умолчанию они содержать сегментный адрес PSP.

2.2. Простейшая программа на ассемблере. Для того, чтобы создать программу в виде исполняемого файла, её предварительно пишут в виде текста, который понятен программисту. Символическое обозначение команд называют «мнемоническим обозначением», или просто «мнемоникой». Приведём пример исходного текста простейшей программы типа EXE на языке ассемблер:

```
dat     segment 'data'
        msg db 'Hello, big world!', 0Dh, 0Ah, '$'
dat     ends
```

```
stk     segment stack 'stack'
        db 512 dup(0)
stk     ends

prog    segment 'code'
        assume CS:prog, DS:dat, SS:stk

begin:  mov AX, dat
        mov DS, AX
        mov DX, offset msg
        mov AH, 09h
        int 21h

        mov AH, 4Ch
        mov AL, 0
        int 21h
prog    ends
end     begin
```

Данная программа выводит на экран надпись «Hello, big world!». Первая строка программы:

```
dat     segment 'data'
```

определяет начало сегмента данных с символическим именем «dat». Далее следует содержимое сегмента данных. В нашем случае в сегменте данных будет содержаться только текст строки, которую нужно вывеси на экран. Выражение:

```
msg db 'Hello, big world!', 0Dh, 0Ah, '$'
```

вводит переменную с именем «msg». Ключевое слово «db» говорит о том, что это будет переменная, состоящая из указанной далее последовательности байт («db» от «define byte» – «определить байт»). Далее идёт сама последовательность байт. Как видно, она может задаваться непосредственно значениями каждого отдельного байта, так и строкой символов (тогда в качестве значений каждого байта будет использован код соответствующего символа). В памяти эта комбинация байт будет выглядеть так:

48	65	6C	6C	6F	2C	20	62	69	67	20	77	6F	72	6C	64	21	0D	0A	24
H	**e**	**l**	**l**	**o**	**,**		**b**	**i**	**g**		**w**	**o**	**r**	**l**	**d**	**!**			**$**

Это наша строка, которую нужно будет вывести. Символ «$» означает конец строки. Следующее выражение:

```
dat     ends
```

определяет конец сегмента данных «dat». Далее следует определение сегмента стека с символическим именем «stk»:

```
stk     segment stack 'stack'
        db 512 dup(0)
stk     ends
```

Выражение:

```
db 512 dup(0)
```

не определяет какой-либо переменной, а просто отводит под стек 512 байт и заполняет их нулями. Далее следует начало сегмента кода с символическим именем «prog»:

```
prog    segment 'code'
```

Выражение:

```
assume CS:prog, SS:stk, DS:dat
```

означает, что регистр CS по умолчанию будет содержать сегментный адрес сегмента кода «prog», а регистр SS – сегментный адрес сегмента стека «stk». Однако последняя часть выражения:

```
DS:dat
```

не задаёт значения регистра DS (он содержит сегмент PSP), а определяет, что при последующем обращении к переменным по имени их следует искать в сегменте данных «dat». Сегментный адрес сегмента «dat» необходимо будет вручную записать в регистр DS. Следующее выражение:

```
begin: mov AX, dat
```

даёт команду процессору скопировать сегментный адрес сегмента «dat» в AX. Слово «begin:» указывает на то, что этому месту программы присваивается символическое имя «begin». Это т.н. «метка». Она понадобится нам в дальнейшем. Затем содержимое AX копируется в DS:

```
mov DS, AX
```

Сегментные регистры обладают одной особенностью: их содержимое нельзя копировать непосредственно из одного в другой. Поэтому пришлось воспользоваться AX в качестве промежуточного регистра. Следующая операция:

```
mov DX, offset msg
```

копирует смещение переменной «msg» в сегменте данных «dat» в регистр DX. Далее в регистр AH копируется байт 0x09:

```
mov AH, 09h
```

Следующая команда:

```
int 21h
```

инициирует вызов прерывания с номером 0x21. Как отмечалось в п.1.4 прерывания – это готовые программы, записываемые в оперативную память BIOS и MS-DOS при загрузке и выполняющие конкретные задачи. Адреса этих программ (точки входа) находятся в таблице векторов прерываний. Каждому прерыванию присваивается условный номер. Точка входа N-го прерывания определяется следующим образом: сегментный адрес хранится по адресу N*4 (слово), смещение по адресу N*4 + 2 (слово). При вызове N-го прерывания процессор выполняет следующие действия:

- сохраняет текущие значения регистров CS и IP в стеке
- считывает из памяти слово по адресу N*4 и загружает его в CS
- считывает из памяти слово по адресу N*4 + 2 и загружает его в IP

Теперь регистры CS:IP указывают на адрес в памяти, по которому располагается N-е прерывание, и процессор начинает выполнять этот машинный код. После выполнения прерывания процессор выполняет следующие действия:

- извлекает значения регистров CS и IP из стека и помещает их в эти регистры
- увеличивает значение IP на 2 (команда вызова прерывания занимает 2 байта)

Т.о. после выполнения прерывания регистры CS:IP указывают на адрес команды, следующей в нашей программе за вызовом прерывания, а значит процессор продолжит выполнение нашей программы[2].

Прерывание 0x21 – это прерывание MS-DOS, и оно может выполнять несколько различных функций. Какую именно функцию будет выполнять прерывание, определяет значение регистра AH (такому правилу подчиняются и другие прерывания). Это значение принято называть функцией вызываемого прерывания. Т.е. в нашем случае мы вызываем функцию 0x09 прерывания 0x21 – вывод строки на экран, которая расположена в памяти по адресу DS:DX. Т.о. наши первые 4 команды – это подготовка к вызову прерывания:

```
begin:  mov AX, dat
        mov DS, AX
        mov AH, 09h
        mov DX, offset msg
```

Символ «$» является для вызываемой функции признаком конца строки, байт 0x0D – означает возврат каретки, 0x0A – перевод строки. Поэтому после вывода строки «Hello, big word!» курсор переместится в начало следующей строки на экране.

Следующие 2 команды также являются подготовкой к вызову прерывания, а 3-я опять вызывает прерывание 0x21:

```
mov AH, 4Ch
mov AL, 0
int 21h
```

Теперь мы вызываем функцию 0x4C прерывания 0x21 – завершение работы программы, освобождение занятой памяти и передача управления MS-DOS. В регистре AL должен находиться код завершения программы. Мы будем использовать код 0 – нормальное завершение. Отличный от нуля код обычно используется в случае завершения программы из-за возникшей ошибки.

Предпоследняя строка программы:

```
prog    ends
```

означает конец сегмента кода. Программа заканчивается выражением:

```
end     begin
```

означающим конец программы и указывающим с помощью метки на то место программы, с которого должно начаться её выполнение (т.е. задающим точку входа). В соответствии с указанной меткой при загрузке программы будет задано начальное значение регистра IP.

Для того чтобы из текста нашей программы получить исполняемый EXE файл необходимо сохранить текст программы в файле с расширением ASM (в MS-DOS для этого проще всего использовать текстовый редактор EDIT.COM, а в Windows – «Блокнот»), а затем воспользоваться программой-транслятором. Мы будем использовать транслятор от Microsoft MASM.EXE и LINK.EXE (последнюю программу называют ещё «компоновщиком»). Файлы MASM.EXE, LINK.EXE и наш ASM-файл необходимо скопировать в одну папку. Для упрощения процедуры трансляции можно создать там BAT-файл (например, MAKE.BAT) со следующим текстом:

```
MASM.EXE %1 %1 0 0
IF NOT ERRORLEVEL 1 LINK.EXE /CO %1, %1;
```

Теперь для выполнения трансляции достаточно из командной строки дать команду:

[2] Именно такому механизму работы прерывания обязаны своим названием.

```
MAKE [имя ASM-файла]
```

Если программа написана без ошибок, то в папке появится EXE-файл с тем же именем, что и ASM-файл. Помимо этого будет создан одноимённый LST-файл, где можно увидеть написанную программу в машинном коде. Для отладки программы можно использовать отладчик CV.EXE, в котором имеется возможность отслеживания изменений всех регистров после выполнения каждой команды. Программу-отладчик CV.EXE необходимо скопировать в папку, где находится EXE-файл программы. Её запуск из командной строки:

```
CV [имя EXE-файла]
```

Образ нашей программы после загрузки EXE-файла в память (например, начиная с адреса 0x0B150) будет выглядеть так:

Адрес	Содержимое	Назначение
0B15:0000h . . .	256 байт	PSP
0B25:0000h 0B25:0010h 0B25:0020h	48 65 6C 6C 6F 2C 20 62 69 67 20 77 6F 72 6C 64 21 0D 0A 24 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	Данные (строка для вывода)
0B27:0000h . . .	512 байт, все равны 0	Стек
0B47:0000h 0B47:0010h	B8 25 0B 8E D8 BA 00 00 B4 09 CD 21 B4 4C B0 00 CD 21	Код

Как видно, сам машинный код занял 18 байт:

Адрес	Содержимое	Мнемоника
0B47:0000h	B8 25 0B	mov AX, dat (0x0B25)
0B47:0003h	8E D8	mov DS, AX
0B47:0005h	BA 00 00	mov DX, offset msg (0x0000)
0B47:0008h	B4 09	mov AH, 09h
0B47:000Ah	CD 21	int 21h
0B47:000Ch	B4 4C	mov AH, 4Ch
0B47:000Eh	B0 00	mov AL, 0
0B47:0010h	CD 21	int 21h

Начальные значения регистров будут:

CS = 0x0B55, IP = 0x0000, SS = 0x0B35, SP = 0x0200, DS = ES = 0xB15

В заключение этого пункта отметим, что в текст программы можно вставлять комментарий. Для этого используют символ «;». Всё, что находится правее этого символа, будет игнорироваться транслятором. Пример:

```
;организовываем выход из программы
mov AH, 4Ch     ;запишем в AH число 0x4C (функция завершения программы)
mov AL, 0       ;запишем в AL число 0x00 (код нормального завершения)
int 21h         ;вызов прерывания 0x21
```

2.3. Практические задания.

- ✓ Построить исполняемый EXE-файл программы на стр. 9–10.
- ✓ Изобразить образ аналогичной программы в памяти, выполненной в виде COM-файла (считать, что программа загружена по адресу 0x0B150).

Новые выражения:

[имя] segment 'code' – начало сегмента кода.

[имя] segment stack 'code' – начало сегмента стека.

[имя] segment 'data' – начало сегмента данных.

[имя] ends – конец сегмента.

[имя] db '[строка]' – объявление массива байт, содержащего строку.

[имя] db [значение], [значение], ... – объявление массива байт, с заданием значения каждого элемента.

[имя] db [число элементов] dup([значение]) – объявление массива байт, с заданием числа его элементов и первоначальным значением всех элементов.

assume CS:[имя сегмента], SS:[имя сегмента], DS:[имя сегмента] – задаёт начальные значения регистров CS, SS, указывает транслятору на то, что при обращении к переменным по имени их следует искать в сегменте, указанном в DS.

mov [приёмник], [источник] – копирование содержимого источника в приёмник. Источником и приёмником могут быть регистры, переменные, ячейки памяти. Источником также может быть имя сегмента (при копировании его сегментного адреса) или само значение (байт или слово), которое необходимо скопировать. Источник и приёмник должны иметь одинаковый размер (байт или слово). Одновременно приёмником и источником не могут быть сегментные регистры, переменные и ячейки памяти.

offset [имя переменной] – получение смещения переменной.

int [номер прерывания] – вызов прерывания.

end [метка] – конец программы с указанием точки входа.

III. Подпрограммы.

3.1. Объявления и вызов подпрограмм. Подпрограммой называют фрагмент программы, выполняющий какую-либо задачу. Подпрограммы создаются с целью фрагментации программы на логически завершённые участки, что облегчает их понимание, отладку и собственно написание. Другая цель создания подпрограмм – получить возможность многократно использовать один и тот же фрагмент кода, когда это необходимо.

Приведём пример подпрограммы, которая выводит на экран строку, адрес которой записан в регистрах DS:DX:

```
print   proc        ;начало подпрограммы с именем «print»
        mov AH, 09h
        int 21h
        ret         ;возврат из подпрограммы
print   endp        ;конец подпрограммы «print»
```

Подпрограмма обязательно должна заканчиваться командой «ret» – возврат из подпрограммы. Вызов такой подпрограммы будет осуществляться командой:

```
call print
```

Приведём пример программы, использующей определённую выше подпрограмму:

```
dat     segment 'data'
        msg1 db 'This is my first program.', 0Dh, 0Ah, '$'
        msg2 db 'The program was completed successfully!', 0Dh, 0Ah, '$'
dat     ends

stk     segment stack 'stack'
        db 512 dup(0)
stk     ends
```

```
prog    segment 'code'
        assume CS:prog, DS:dat, SS:stk

begin:  mov AX, dat
        mov DS, AX

        mov DX, offset msg1
        call print

        mov DX, offset msg2
        call print

        mov AH, 4Ch
        mov AL, 0
        int 21h

print   proc
        mov AH, 09h
        int 21h
        ret
print   endp

prog    ends
end     begin
```

Процесс вызова подпрограммы подобен процессу вызова прерывания. При получении команды «call» процессор:

- помещает в стек смещение следующей команды, «точку возврата» (значение IP + 3, т.к. сама команда «call» занимает 3 байта)
- в IP помещает смещение начала подпрограммы
- выполняет подпрограмму до получения команды «ret»

При получении команды «ret» процессор:

- извлекает из стека значение точки возврата и помещает его в регистр IP

В отличие от вызова прерывания при вызове подпрограммы командой «call» значение CS не меняется, т.к. основная программа и подпрограмма находятся в одном сегменте кода.

Образ приведённой выше программы после загрузки EXE-файла в память (снова возьмём начальный адрес 0x0B150) будет выглядеть так:

Адрес	Содержимое	Назначение
0B15:0000h . . .	256 байт	PSP
0B25:0000h 0B25:0010h 0B25:0020h 0B25:0030h 0B25:0040h	54 68 69 73 20 69 73 20 6D 79 20 66 69 72 73 74 20 70 72 6F 67 72 61 6D 2E 0D 0A 24 54 68 65 20 70 72 6F 67 72 61 6D 20 77 61 73 20 63 6F 6D 70 6C 65 74 65 64 20 73 75 63 63 65 73 73 66 75 6C 6C 79 21 0D 0A 24 00 00 00 00 00 00 00 00 00 00	Данные (строки для вывода)
0B2A:0000h . . .	512 байт, все равны 0	Стек
0B4A:0000h 0B4A:0010h	B8 25 0B 8E D8 BA 00 00 E8 0C 00 BA 1C 00 E8 06 00 B4 4C B0 00 CD 21 B4 09 CD 21 C3	Код

Машинный код занял 28 байт:

Адрес	Содержимое	Мнемоника	
0B4A:0000h	B8 25 0B	mov AX, dat	Основная программа
0B4A:0003h	8E D8	mov DS, AX	
0B4A:0005h	BA 00 00	mov DX, offset msg1 (0x0000)	
0B4A:0008h	E8 0C 00	call print (0x000C)	
0B4A:000Bh	BA 1C 00	mov DX, offset msg2 (0x001C)	
0B4A:000Eh	E8 06 00	call print (0x0006)	
0B4A:0011h	B4 4C	mov AH, 4Ch	
0B4A:0013h	B0 00	mov AL, 0	
0B4A:0015h	CD 21	int 21h	
0B4A:0017h	B4 09	mov AH, 09h	Подпрограмма «print»
0B4A:0019h	CD 21	int 21h	
0B4A:001Bh	C3	ret	

Начальные значения регистров будут:

CS = 0x0B4A, IP = 0x0000, SS = 0x0B2A, SP = 0x0200, DS = ES = 0x0B15

Заметим, что в тексте программы операндом команды «call» является имя подпрограммы. При трансляции программы в машинный код это имя заменяется смещением начала подпрограммы, но не от начала сегмента, а относительно точки возврата. Это смещение рассматривается как число со знаком и может быть отрицательным, если подпрограмма располагается перед точкой входа основной программы.

3.2. Практические задания.

- ✓ Написать программу аналогичную программе на стр. 14–15, которая для вывода строки на экран использует подпрограмму «print», а для завершения своей работы подпрограмму «exit».
- ✓ Изобразить образ аналогичной программы в памяти, выполненной в виде COM-файла (условия те же, что и п.2.3).

Новые выражения:

`[имя] proc` – начало подпрограммы.
`[имя] endp` – конец подпрограммы.
`call [имя подпрограммы]` – вызов подпрограммы.

IV. Циклы.

4.1. Организация циклов. В некоторых случаях использование подпрограмм для многократного выполнения какого-либо участка кода становится нецелесообразным, т.к. приходится слишком часто осуществлять их вызов. В таких случаях удобно организовать циклическое выполнение нужного фрагмента программы. Рассмотрим конкретный пример:

```
        mov CX, 30
symbol: mov AH, 02h
        mov DL, '*'
        int 21h
        loop symbol
```

Строки 2 – 4 осуществляют вызов DOS-прерывания 0x21, используя функцию 0x02. Эта функция выводит на экран символ, код которого записан в DL (в нашем примере звёздочка).

Затем следует выражение:

```
loop symbol
```

При получении команды «loop» процессор уменьшит на единицу значение регистра CX. Если новое значение CX отлично от нуля, то в IP будет записано смещение команды, соответствующей метке, указанной в качестве операнда команды «loop» (в нашем случае это метка «symbol»). Т.о. процессор снова перейдёт к выполнению строки 2. После выполнения строк 2, 3, 4 на экран снова будет выведена звёздочка. Затем опять будет выполнена команда «loop» и т.д. Когда же при очередном выполнении команды «loop» значение CX станет равным 0, процессор просто перейдёт к выполнению следующей команды. Т.о. регистр CX при организации циклов играет роль счётчика, и перед началом цикла в него записывают число повторений следующего далее фрагмента кода. В нашем примере перед началом цикла в CX было записано число 30, следовательно, на экран будет выведена строка из 30 звёздочек.

4.2. Использование стека. Часто регистр CX приходится использовать для различных целей внутри циклически повторяющегося фрагмента кода. Для того, чтобы такой цикл был правильно выполнен, значение регистра CX необходимо каждый раз при входе в повторяющийся фрагмент где-то сохранять, а перед вызовом «loop» восстанавливать. Одним из решений является использование стека:

```
        mov CX, 30
cycle:  push CX      ;сохранение значения CX в стеке
        ...          ;выполнение каких-либо действий,
        ...          ;при которых значение CX будет изменено
        pop CX       ;извлечение слова из стека и сохранение его в CX
        loop cycle
```

При получении команды «push CX» процессор помещает в стек слово из регистра CX. После выполнения операций, в ходе которых значение регистра CX менялось, непосредственно перед командой «loop» из стека извлекается сохранённое слово и помещается в CX (команда «pop CX»).

4.3. Использование других регистров в качестве счётчиков. Другим, на первый взгляд очевидным решением описанной в п.4.2 проблемы является использование другого регистра в качестве счётчика. Однако команды, аналогичной «loop», но использующей другой регистр в качестве счётчика в машинном коде 8086 нет. Единственный выход – вручную осуществлять изменение счётчика и определять, был ли повторён нужный фрагмент определённое число раз, или нет:

```
        mov SI, 50
cycle:  ...
        ...
        dec SI       ;уменьшаем SI на 1
        cmp SI, 0    ;сравниваем SI с 0
        jnz cycle    ;если флаг нуля не установлен, возвращаемся на
                     ;метку «cycle»
```

Как видно, в приведённом примере в качестве счётчика использовался регистр SI. При получении команды:

```
dec SI
```

процессор уменьшает значение SI на единицу. Затем следует выражение:

```
cmp SI, 0
```

означающая, что необходимо выполнить сравнение значения SI со следующим в качестве операнда числом. Процессор вычитает из значения SI это число и, если результат равен нулю, устанавливает флаг нуля. В противном случае флаг нуля сбрасывается. Следующая команда:

```
jnz cycle
```

записывает в IP смещение команды, соответствующей метке, указанной в качестве операнда. Причём эта операция выполняется, только если флаг нуля сброшен. В противном случае эта команда игнорируется и выполняется следующая за ней команда. Т.о. до тех пор, пока значение SI не станет равным нулю, выполнение программы будет возвращаться на метку «cycle» командой «jnz». Поскольку в SI в нашем примере было записано число 50, и каждый раз перед сравнением с нулём значение SI уменьшается на единицу, строки 2 – 5 будут выполнены 50 раз.

Отметим, что кроме «jnz» часто используется команда «jz»:

```
jz [метка]
```

Она работает аналогично «jnz», но выполняется, если флаг нуля установлен.

Аналогичный пример:

```
        mov SI, 0
cycle:  ...
        ...
        ...
        inc SI        ;увеличиваем SI на 1
        cmp SI, 50    ;сравниваем SI с 50
        jnz cycle     ;если флаг нуля установлен, возвращаемся на
                      ;метку «cycle»
```

Здесь при выполнении цикла SI наоборот увеличивается от нуля до 50. Команда:

```
inc SI
```

увеличивает значение SI на единицу. Команды «dec» и «inc» называют соответственно «декрементом» и «инкрементом», а команды «jnz» и «jz» – «командами условного перехода». Существует несколько разновидностей команд условного перехода, в зависимости от того какой из флагов они анализируют при выполнении. В дополнение отметим ещё две пары подобных команд: «js» и «jns», «jc» и «jnc». Они работают аналогично предыдущей паре команд, но «js» и «jns» анализируют флаг знака, а пара команд «jc» и «jnc» – флаг переноса.

Вместо команд инкремента и декремента можно пользоваться командами сложения и вычитания:

```
add SI, 1
sub SI, 1
```

Первая команда добавляет к содержимому SI единицу, а вторая вычитает. Результат сохраняется в регистре SI.

4.4. Практические задания.

- ✓ Написать программу, выводящую на экран надпись:
  ```
  --------------------------------------------------
  Hello, big world!
  ==================================================
  ```
  Для вывода строк должны использоваться подпрограммы, причём первую и третью строку должна выводить одна подпрограмма использующая цикл с командой «loop».
- ✓ Выполнить предыдущее задание, используя вместо команды «loop» команду «jnz».

Новые выражения:

loop [метка] – уменьшает значение CX на единицу, если новое значение CX отлично от нуля, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

push [источник] – помещает слово из источника в стек. Источником может быть 16-разрядный регистр, ячейка памяти, переменная (слово) или само помещаемое значение.

pop [приёмник] – извлекает слово из стека и помещает его в приёмник. Приёмником может быть 16-разрядный регистр кроме CS, ячейка памяти или переменная (слово).

inc [источник] – увеличивает на единицу значение источника. Источником могут быть регистры (кроме сегментных), переменная или ячейка памяти.

dec [источник] – уменьшает на единицу значение источника. Источником могут быть регистры (кроме сегментных), переменная или ячейка памяти.

add [приёмник], [источник] – складывает значение источника и приёмника и сохраняет результат в приёмнике. Источником и приёмником могут быть регистры (кроме сегментных), переменные или ячейки памяти. Источником также может быть само значение (байт или слово), которое складывается с приёмником. Источник и приёмник одновременно не могут быть переменными или ячейками памяти.

sub [приёмник], [источник] – вычитает из значения приёмника значение источника и сохраняет результат в приёмнике. Источником и приёмником могут быть регистры (кроме сегментных), переменные или ячейки памяти. Источником также может быть само значение (байт или слово), которое вычитается из приёмника. Источник и приёмник одновременно не могут быть переменными или ячейками памяти.

cmp [источник-1], [источник-2] – сравнивает значения источника-1 и источника-2 путём вычитания 2-го из 1-го. В соответствии с результатом вычитания устанавливает флаг нуля. Источниками могут быть регистры (кроме сегментных), переменные или ячейки памяти. Источником-2 может также быть само значение (байт или слово), с которым сравнивается источник-1. Оба источника должны иметь одинаковый размер (байт или слово). Оба источника одновременно не могут быть переменными или ячейками памяти.

jnz [метка] – если флаг нуля сброшен, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

jz [метка] – если флаг нуля установлен, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

jns [метка] – если флаг знака сброшен, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

js [метка] – если флаг знака установлен, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

jnc [метка] – если флаг переноса сброшен, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

jc [метка] – если флаг переноса установлен, изменяет IP так, что программа выполняется с указанной метки, в противном случае выполняется следующая команда.

V. Ввод с клавиатуры.

5.1. Ввод с помощью прерывания MS-DOS. Для организации ввода с клавиатуры удобно воспользоваться функцией 0x08 DOS-прерывания 0x21. Приведём пример программы, которая ждёт, пока пользователь не введёт 3 символа с клавиатуры, а затем выводит их в виде строки:

```
dat     segment 'data'
        buff db 4 dup(0)   ;буфер, куда будем помещать вводимые символы
dat     ends

stk     segment stack 'stack'
        db 512 dup(0)
```

```
stk     ends

prog    segment 'code'
        assume CS:prog, DS:dat, SS:stk

begin:  mov AX, dat
        mov DS, AX

        mov BX, offset buff    ;DS:BX указывают на 1-й байт буфера

        mov AH, 08h
        int 21h
        mov [BX], AL    ;копируем содержимое AL в буфер (по адресу DS:BX)

        inc BX          ;DS:BX указывают на 2-й байт буфера

        mov AH, 08h
        int 21h
        mov [BX], AL    ;копируем содержимое AL в буфер (по адресу DS:BX)

        inc BX          ;DS:BX указывают на 3-й байт буфера

        mov AH, 08h
        int 21h
        mov [BX], AL    ;копируем содержимое AL в буфер (по адресу DS:BX)

        inc BX          ;DS:BX указывают на 4-й байт буфера

        mov AL, '$'
        mov [BX], AL    ;копируем символ '$' в конец буфера

        mov AH, 09h
        mov DX, offset buff
        int 21h

        mov AH, 4Ch
        mov AL, 0
        int 21h
prog    ends
end     begin
```

При вызове функции 0x08 DOS-прерывания 0x21 процессор переходит в режим ожидания, пока не будет нажата клавиша на клавиатуре. После этого код нажатого символа (байт) записывается в регистр AL (код клавиши ESC – 0x1B, ENTER – 0x0D)[3]. После ввода каждого символа, код символа пересылается в буфер:

```
mov [BX], AL
```

Квадратные скобки указывают на то, что в качестве приёмника используется не сам регистр BX, а ячейка памяти с адресом DS:BX. Вместо BX можно использовать также регистры SI и DI. Для того, чтобы 1-й символ переслать в 1-й байт буфера, 2-й символ во 2-й байт и т.д. значение BX увеличивается на единицу перед каждым вводом. Чтобы вывести на экран полученную из введённых символов строку с помощью функции 0x09 DOS-прерывания 0x21 необходимо в конец буфера записать символ «$».

[3] В Windows для получения кодов символов можно воспользоваться программой «Таблица символов».

5.2. Практические задания.

- ✓ Написать программу, выводящую на экран сообщение «Enter string: », а затем предлагающая пользователю ввести строку символов (макс. 50). При этом после нажатия клавиши символ должен сразу выводится на экран. Если в процессе ввода пользователь нажмёт ENTER или введёт 50 символов, то программа должна вывести на экран сообщение «Your string: » и далее введённая пользователем строка, после чего завершить работу. Если в процессе ввода пользователь нажмёт ESC, то программа должна вывести на экран сообщение «Process was terminated!» и завершить свою работу.

VI. Работа с файлами.

6.1. Создание и открытие файлов. Прерывание 0x21 MS-DOS предоставляет несколько функций для различных операций с файлами: создание, открытие, запись, чтение и др. Рассмотрим сначала пример создания файла:

```
dat     segment 'data'
        filename db 'MYFILE.TXT', 0          ;имя создаваемого файла
        msgOK db 'File was created!', 0Dh, 0Ah, '$'
        msgERR db 'Cannot to create file!', 0Dh, 0Ah, '$'
        handle dw 0                          ;дескриптор файла
dat     ends

stk     segment stack 'stack'
        db 512 dup(0)
stk     ends

prog    segment 'code'
        assume CS:prog, DS:dat, SS:stk

begin:  mov AX, dat
        mov DS, AX

        mov DX, offset filename
        mov AH, 3Ch
        mov CX, 0
        int 21h

        jc fileER;

        mov handle, AX

        mov AH, 09h
        mov DX, offset msgOK
        int 21h

        mov BX, handle
        mov AH, 3Eh
        int 21h

        jmp exit

fileER: mov AH, 09h
        mov DX, offset msgERR;
        int 21h

exit:   mov AH, 4Ch
        mov AL, 0
```

```
        int 21h
prog    ends
end     begin
```

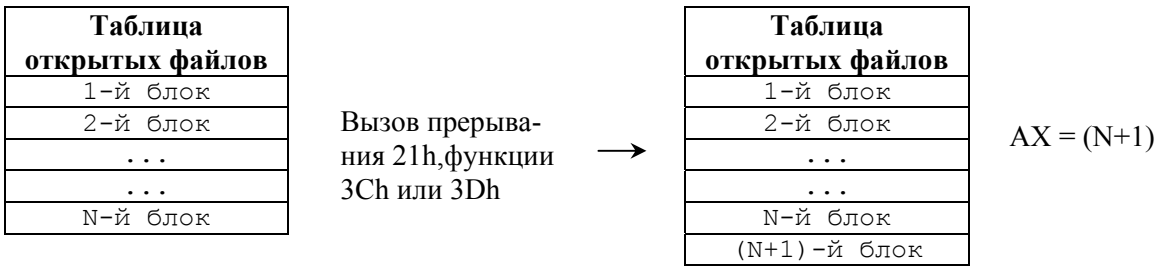
Для создания файла используется функция 0x3C DOS-прерывания 0x21:

```
mov DX, offset filename
mov AH, 3Ch
mov CX, 0
int 21h
```

При её вызове в регистре CX должны находиться атрибуты файла (0x01 – файл только для чтения, 0x02 – скрытый файл, 0x04 – системный файл, 0x20 – архивный файл), а в DS:DX – адрес строки с именем файла, заканчивающейся нулевым байтом. При этом можно задать вместе с именем файла и путь, например, «C:\WORK\MYFILE.TXT». Если будет задано только имя файла, то он будет создан в текущем каталоге (по умолчанию это тот каталог, из которого была запущена наша программа). Если файл с заданным именем уже существует, то его размер усекается до нуля и все данные в нём теряются, после чего файл открывается.

При успешном создании файла в AX записывается его дескриптор (descriptor – «описатель»). В случае ошибки устанавливается флаг переноса.

Когда DOS создаёт или открывает файл, в специальной области оперативной памяти создаётся т.н. «блок описания файла». В нём хранится разнообразная информация, необходимая MS-DOS для работы с файлом (имя файла, его атрибуты, информация о физическом положении файла на диске и др.). Каждый такой блок занимает 59 байт. MS-DOS группирует эти блоки в «таблицу открытых файлов». Дескриптор же представляет собой 16-битный номер блока описания файла в таблице открытых файлов:



Все дальнейшие операции с файлом осуществляются с использованием дескриптора. Т.о. понятие «открытый файл» можно определить так – это файл, для которого создан блок описания в таблице открытых файлов.

В нашем примере создаётся файл с именем «MYFILE.TXT». Для простоты файлу не присваивается каких-либо атрибутов (CX = 0). Если файл не удалось создать (установлен флаг переноса), то с помощью команды «jc» выполнение программы переходит на метку «fileER» и выводится сообщение об ошибке «Cannot to create file!», после чего программа завершает работу. Если файл был создан (сброшен флаг переноса), то дескриптор файла сохраняется в переменной «handle»:

```
mov handle, AX
```

Поскольку дескриптор – 16-битное число, то переменная «handle» определена как слово («dw» от «define word» – «определить слово»):

```
handle dw 0
```

Далее программа выводит сообщение об успешном создании файла «File was created!». После того как операции с открытым файлом завершены, его необходимо закрыть. Оставлять файлы открытыми после выполнения программы не рекомендуется, т.к. это может вызвать потерю данных или

повреждение логической структуры диска. Для закрытия файла используется функция 0x3E DOS-прерывания 0x21:

```
mov BX, handle
mov AH, 3Eh
int 21h
```

При её вызове в регистре BX должен находиться дескриптор открытого файла. После закрытия файла выполнение нашей программы переходит на метку «exit» благодаря команде:

```
jmp exit
```

Эта команда выполняется также как и команды условного перехода, с той разницей, что выполняется всегда, вне зависимости от состояния каких-либо флагов.

Для того чтобы открыть уже существующий файл без уничтожения его содержимого, можно воспользоваться функцией 0x3D DOS-прерывания 0x21. При её вызове в регистре AL должны находиться код доступа к файлу (0x00 – открыть только для чтения, 0x01 – для записи, 0x02 – для чтения и записи), а в DS:DX – адрес строки с именем файла, заканчивающейся нулевым байтом. При успешном открытии файла в AX записывается его дескриптор. В случае ошибки (в т.ч. если указанный файл не существует) устанавливается флаг переноса.

6.2. Чтение и запись в файл. Для чтения и записи данных в файл используют функции 0x3F, 0x40 и 0x42 DOS-прерывания 0x21. Все функции требуют наличия в регистре BX дескриптора файла.

Функция 0x3F читает из файла число байт, указанное в регистре CX и помещает его в буфер по адресу DS:DX. Функция 0x40 записывает в файл число байт, указанное в регистре CX из буфера по адресу DS:DX. При ошибке обе функции устанавливают флаг переноса.

Чтение или запись осуществляется с того байта в файле, который задан в т.н. «указателе файла». Указатель файла хранится в блоке описания файла. По умолчанию после создания или открытия файла он равен нулю и т.о. указывает на начало файла. После успешного чтения или записи N байт указатель автоматически увеличивается на N байт. Т.о. следующее чтение или запись начнётся с N-го байта в файле.

Значение указателя файла можно принудительно изменить функцией 0x42. При её вызове в AL должен быть записан режим установки указателя (0x00 – смещение от начала файла, 0x01 – смещение от текущей позиции указателя, 0x02 – смещение от конца файла). Само смещение задаётся 32-битным числом, и поэтому записывается в два регистра (в CX – старшее слово, в DX – младшее слово). Если AL = 0x01 или 0x02, то смещение рассматривается как число со знаком (т.е. может быть отрицательным). После вызова этой функции указатель файла перемещается на величину, равную заданному смещению (вперёд или назад и относительно какой позиции, определяет регистр AL). Новое значение указателя сохраняется в регистрах DX и AX (DX – старшее слово, AX – младшее слово).

Пример программы, которая дописывает в конец файла две строки «12345» и «abcde»:

```
dat     segment 'data'
        filename db 'MYFILE.TXT', 0
        handle dw 0
        str1 db '12345', 0Dh, 0Ah
        str2 db 'abcde', 0Dh, 0Ah
        msgERR db 'Cannot to open file!', 0Dh, 0Ah, '$'
dat     ends

stk     segment stack 'stack'
        db 512 dup(0)
stk     ends

prog    segment 'code'
        assume CS:prog, DS:dat, SS:stk
```

```
begin:  mov AX, dat
        mov DS, AX

        mov DX, offset filename
        mov AH, 3Dh
        mov CX, 1
        int 21h

        jc fileER;

        mov handle, AX

        mov BX, handle
        mov DX, 0
        mov CX, 0
        mov AH, 42h
        mov AL, 2h
        int 21h

        mov BX, handle
        mov DX, offset str1
        mov CX, 7
        mov AH, 40h
        int 21h

        mov BX, handle
        mov DX, offset str2
        mov CX, 7
        mov AH, 40h
        int 21h

        mov BX, handle
        mov AH, 3Eh
        int 21h

        jmp exit

fileER: mov AH, 09h
        mov DX, offset msgERR;
        int 21h

exit:   mov AH, 4Ch
        mov AL, 0
        int 21h
prog    ends
end     begin
```

Как видно, программа открывает файл с именем «MYFILE.TXT». Если файл не удалось открыть, то выдаётся сообщение об ошибке «Cannot to open file!», и программа завершает работу. После открытия файла его дескриптор сохраняется в переменной «handle». Далее программа устанавливает указатель файла в конец файла, для того чтобы осуществить именно дописывание данных в файл:

```
mov BX, handle
mov DX, 0        ;задаём смещение равное 0
mov CX, 0
mov AH, 42h
mov AL, 2h       ;сместим указатель на 0 байт от конца файла
int 21h
```

Далее осуществляется запись строки «srt1» в файл:

```
mov BX, handle
mov DX, offset str1
mov CX, 7            ;записываем 7 байт (длина строки)
mov AH, 40h
int 21h
```

После этого аналогично записывается строка «str2». Поскольку после первой операции записи указатель автоматически переместится вперёд на 7 байт, нет необходимости устанавливать его вручную перед следующей записью. После выполнения всех операций программа закрывает файл и завершает работу.

6.3. Практические задания.

- ✓ Написать программу, выводящую на экран сообщение «Enter file name: », а затем предлагающая пользователю ввести имя файла (макс. 60 символов). Условия ввода должны быть такими же, как в задании п.5.2 (вводимые символы сразу выводятся на экран, ESC – выход с выводом сообщения «Process was terminated!»). При нажатии ENTER или вводе 60 символов программа должна открыть файл с введённым именем, или создать его, в случае отсутствия. Если ни открыть, ни создать файл не удалось, то программа должна завершить работу с выводом сообщения «Cannot to open/create file!». Если файл был открыт или создан, программа должна вывести на экран сообщение «Enter string: » и предложить ввести строку символов (макс. 30). Условия ввода как в предыдущем случае. При нажатии ENTER или вводе 30 символов введённая строка должна быть добавлена в файл.
- ✓ Написать программу, аналогичную предыдущей, которая после ввода имени файла открывает его, и выводит на экран все имеющиеся в нём строки. Размер файла не должен превышать 10 Кб, в противном случае, после его открытия программа должна завершить работу с выводом сообщения «File is too large!».

Новые выражения:

`[имя] dw [значение], [значение], ...` – объявление массива слов, с заданием значения каждого элемента.

`[имя] dw [число элементов] dup([значение])` – объявление массива слов, с заданием числа его элементов и первоначальным значением всех элементов.

`jmp [метка]` – изменяет IP так, что программа выполняется с указанной метки.

VII. Работа с графикой.

7.1. Общие принципы работы с графикой. Для работы с графикой необходимо воспользоваться прерываниями BIOS, т.к. прерывания MS-DOS предоставляют функции только для вывода текстовой информации. Для того чтобы нарисовать на экране простейшее графическое изображение необходимо сначала перевести видеоадаптер в графический режим. До сих пор мы пользовались текстовым режимом, в котором экран условно разбивается на определённое число т.н. «знакомест». Знакоместо – это область экрана, на которую можно вывести только один символ. Стандартное разрешение экрана в текстовом режиме 80x25 (т.е. 80 символов по горизонтали и 25 по вертикали). Т.о. построение изображения в текстовом режиме программно осуществляется по символам. В графическом режиме экран разбивается на «пиксели» – небольшие участки в виде точек, и построение изображения можно осуществлять, рисуя каждую из них в отдельности.

Для перевода видеоадаптера в графический режим можно воспользоваться функцией 0x00 прерывания BIOS 0x10. При её вызове в регистре AL должен быть записан код видеорежима (мы будем использовать код 0x13 – графический режим 320x200, 256 цветов).

Функция 0x0F прерывания BIOS 0x10 записывает код текущего видеорежима в регистр AL. Отметим, что грамотно построенная программа перед изменением видеорежима, должна запомнить код текущего видеорежима и перед завершением работы восстановить его.

Для рисования отдельных пикселей мы будем пользоваться функцией 0x0C прерывания BIOS 0x10. При её вызове в регистре AL должен находиться код цвета рисуемого пикселя, в регистре CX – x-координата, DX – y-координата, а в регистре BH – видеостраница, на которой осуществляется рисование.

Как отмечалось в п.1.4, образ того, что в данный момент изображено на экране, хранится в специальной области оперативной памяти – видеобуфере. Чтобы изменить текущее изображение (вывести строку, символ, нарисовать пиксель), необходимо соответствующим образом изменить содержимое видеобуфера (что и делают используемые нами функции 0x02, 0x09 DOS-прерывания 0x21 и функция 0x0C прерывания BIOS 0x10). Видеобуфер организован так, что в нём можно сохранить сразу несколько «образов» экрана. Каждый такой «образ» называют видеостраницей (другими словами это просто определённая область видеобуфера). Видеостраница, которая в данный момент отображается на экране, называется активной. Такая организация видеобуфера позволяет сначала нарисовать изображение на неактивной видеостранице (т.е. изменения в этой части видеобуфера не будут сразу отображаться на экране), а затем сделать эту страницу активной. В этом случае нарисованное изображение появится на экране мгновенно. Если же осуществлять рисование на активной видеостранице, то при рисовании крупных объектов будет заметен процесс высвечивания отдельных пикселей. Для простоты мы будем пользоваться только одной видеостраницей.

Ниже приведён пример программы, рисующей на экране горизонтальную линию начиная с точки (0, 10) длиной 256 пикселей, причём каждый пиксель линии имеет свой цвет:

```
dat     segment 'data'
        vmode db 0          ;исходный видеорежим
        color db 0          ;код цвета
dat     ends

stk     segment stack 'stack'
        db 512 dup(0)
stk     ends

prog    segment 'code'
        assume CS:prog, DS:dat, SS:stk

begin:  mov AX, dat
        mov DS, AX

        mov AH, 0Fh
        int 10h
        mov vmode, AL

        mov AH, 0
        mov AL, 13h
        int 10h

        mov CX, 0
        mov DX, 10
paint:  mov AH, 0Ch
        mov AL, color
        mov BH, 0
        int 10h

        inc CX
```

```
        inc color
        cmp CX, 256
        jnz paint

waitkb: mov AH, 08h
        int 21h
        cmp AL, 1Bh
        jnz waitkb

        mov AH, 0
        mov AL, vmode
        int 10h

        mov AH, 4Ch
        mov AL, 0
        int 21h
prog    ends
end     begin
```

После вычерчивания линии программа ожидает нажатия клавиши ESC, после чего восстанавливает исходный видеорежим и завершает работу.

7.2. Практические задания.

- ✓ Написать программу, выводящую на экран изображение российского флага.