

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ  
«ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ-УЧЕБНО-НАУЧНО-  
ПРОИЗВОДСТВЕННЫЙ КОМПЛЕКС»

О.В. Конюхова  
Э.А. Кравцова

**Программное обеспечение вычислительных машин и систем.  
Программирование на языке ассемблера**

Рекомендовано ФГБОУ ВПО «Госуниверситет - УНПК» для  
использования в учебном процессе в качестве практикума для высшего  
профессионального образования

Орел, 2014

УДК  
ББК  
К

Рецензенты:

кандидат технических наук, доцент кафедры «Информационные системы» Федерального государственного образовательного учреждения высшего профессионального образования «Государственный университет – учебно-научно-производственный комплекс»

*А.П. Гордиенко,*

доктор физико-математических наук, профессор кафедры «Математическое моделирование» Федерального государственного образовательного учреждения высшего профессионального образования «Тульский государственный университет»

*В.И. Желтков*

**Конюхова, О.В., Кравцова, Э.А.**

Программное обеспечение вычислительных машин и систем. Программирование на языке ассемблера/ О.В. Конюхова, Э.А. Кравцова. – Орел: ФГБОУ ВПО «Гос-университет-УНПК», 2014. – 130 с.

К Практикум содержит краткие теоретические сведения по программированию на языке ассемблера основных алгоритмических конструкций и структур данных; программированию на языке ассемблера устройств (клавиатуры, мыши, системных часов, дисплея и т.п.) фон-неймановских ВМ, а также перечень заданий к лабораторным работам и практическим занятиям, соответственно.

Практикум соответствует требованиям ФГОС и содержанию программ учебных дисциплины «Архитектура ЭВМ и систем» для студентов направлений 09.03.02 «Информационные системы и технологии», 09.03.04 «Программная инженерия», 09.03.01 «Информатика и вычислительная техника»; дисциплины «Вычислительные системы, сети и телекоммуникации» для студентов направления 09.03.03 «Прикладная информатика».

Предназначен для студентов направлений 09.03.02 «Информационные системы и технологии», 09.03.04 «Программная инженерия», 230700.62 «Прикладная информатика», 09.03.01 «Информатика и вычислительная техника», 09.03.03 «Прикладная информатика». Может быть полезен студентам других специальностей при изучении информатики и низкоуровневого программирования.

УДК  
ББК

© ФГБОУ ВПО «Госуниверситет-УНПК», 2014

## Содержание

ВВЕДЕНИЕ	6
1. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА	7
1.1. Организация памяти для хранения программ	7
1.2. Регистры процессора Intel 8086	8
1.3. Вычисление физического адреса в процессоре Intel 8086	12
1.4. Классификация и структура команд процессора	13
1.5. Способы адресации данных процессора Intel 8086	18
1.6. Способы адресации команд процессора Intel 8086	25
1.7. Общий формат ассемблерной команды	29
1.8. Определение данных	30
1.9. Основные команды языка ассемблера	31
2. ПРОГРАММИРОВАНИЕ УСТРОЙСТВ НА ЯЗЫКЕ АССЕМБЛЕРА	62
2.1. Прерывания, исключения и механизм их обработки	62
2.2. Организация работы с файлами	65
2.3. Управление клавиатурой	79
2.4. Управление выводом информации на дисплей	84
2.5. Управление таймером	90
2.6. Управление прерываниями. Написание собственного прерывания	94
2.7. Управление мышью	95
3. ПРОЦЕСС АССЕМБЛИРОВАНИЯ И ВЫПОЛНЕНИЯ ПРОГРАММЫ	98
3.1. Получение исполняемого файла	98
3.2. Работа в отладчике Turbo Debugger	102
4. ЛАБОРАТОРНЫЙ ПРАКТИКУМ	105
4.1. Общий порядок выполнения лабораторных работ	105
4.2. Лабораторная работа № 1. Линейное исполнение программ. Арифметические и поразрядные логические операции над целыми двоичными числами	106

4.3. Лабораторная работа № 2. Организация межсегментных переходов	108
4.4. Лабораторная работа № 3. Команды условного и безусловного переходов. Организация ветвлений и циклов в программе	110
4.5. Лабораторная работа № 4. Обработка массивов. Числа Фибоначчи	111
4.6. Лабораторная работа № 5. Использование подпрограмм. Сортировка массива чисел	112
4.7. Лабораторная работа № 6. Обработка структур. Ведение базы данных о пациентах	114
4.8. Лабораторная работа № 7. Использование стека. Проверка баланса расстановки скобок в строке	115
4.9. Лабораторная работа № 8. Использование стека и рекурсивных процедур. Организация передачи параметров через стек в процедуру вычисления факториала числа	117
5. ЗАДАНИЯ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ	119
5.1. Общий порядок выполнения практических заданий	119
5.2. Практическое занятие № 1. Управление дисплеем. Вывод символов ASCII на экран	120
5.3. Практическое занятие № 2. Управление дисплеем в графическом режиме	120
5.4. Практическое занятие № 3. Управление клавиатурой. Проверка символа в буфере клавиатуры	121
5.5. Практическое занятие № 4. Управление клавиатурой. Ввод строки символов	122
5.6. Практическое занятие № 5. Файлы последовательного доступа. Запись и чтение информации	123
5.7. Практическое занятие № 6. Файлы прямого доступа. Запись и чтение информации	123
5.8. Практическое занятие № 7. Управление дисками. Организация поиска каталогов и файлов	124

5.9. Практическое занятие № 8. Управление мышью	125
5.10. Практическое занятие № 9. Управление прерываниями. Написание собственного прерывания	126
5.11. Практическое занятие № 10. Управление счётчиком времени суток	126
5.12. Практическое занятие № 11. Генерация звука	127
ЛИТЕРАТУРА	128

## **ВВЕДЕНИЕ**

Вычислительные системы в своем развитии достигли высокого уровня совершенства. Они компактны, обладают большой скоростью выполнения заданий и достаточно просты в обращении. Все эти качества привели к их широкому использованию. Для полного учета всех преимуществ и ограничений, характеризующих процесс решения задачи с помощью вычислительных систем, необходимо знание принципов построения и функционирования как вычислительных систем в целом, так и отдельных их устройств. Для эффективного применения вычислительных машин также необходимо понимание возможностей и знание внутренней структуры современных персональных компьютеров. Основы организации архитектуры вычислительных систем необходимы для знания многих дисциплин.

Практикум состоит из пяти глав. Две первые главы посвящены изложению теоретического материала по программированию на языке ассемблера основных алгоритмических конструкций и структур данных; программированию устройств (клавиатуры, мыши, системных часов, дисплея и т.п.) фон-неймановских ВМ на языке ассемблера. В третьей главе приводится подробное описание процесса ассемблирования программы с применением виртуальной машины DOSBox. Четвёртая и пятая главы содержат перечень заданий к лабораторным работам и практическим занятиям, соответственно.

Выполнение заданий практикума позволит студентам получить достаточно подробное представление о принципах функционирования основных блоков и устройств фон-неймановских ВМ, а так же первичные навыки низкоуровневого программирования на языке ассемблера, что актуально для разработки качественного программного обеспечения.

Практикум соответствует требованиям ФГОС и содержанию программ учебных дисциплины «Архитектура ЭВМ и систем» для студентов направлений 09.03.02 «Информационные системы и технологии», 09.03.04 «Программная инженерия», 09.03.01 «Информатика и вычислительная техника»; дисциплины «Вычислительные системы, сети и телекоммуникации» для студентов направления 09.03.03 «Прикладная информатика». Может быть полезен студентам других специальностей при изучении информатики и низкоуровневого программирования.

# 1. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

## 1.1. Организация памяти для хранения программ

Согласно принципам Джона фон Неймана, вычислительная машина (ВМ) выполняет вычисления в соответствии с программой, которая располагается в памяти ВМ. Любая программа включает в себя команды (операторы) и данные (операнды). Программа выполняется с целью получения результирующих данных на основе преобразования исходных, с возможным формированием промежуточных данных.

В соответствии с концепцией хранимой в памяти программы, и команды и данные располагаются в единой памяти в двоичных кодах. Память представляет собой набор ячеек, каждая из которых имеет свой уникальный номер – *адрес*. Команды и данные хранятся в ячейках, и их местоположение в памяти определяется адресами соответствующих ячеек. Поскольку команды и данные на уровне кодов неотличимы друг от друга, то для различия команд и данных используется их размещение в различных областях памяти – *сегментах*.

**Сегмент** – это прямоугольная область памяти, характеризующаяся начальным адресом и длиной.

**Начальный адрес (адрес начала сегмента)** – это номер (адрес) ячейки памяти, с которой начинается сегмент.

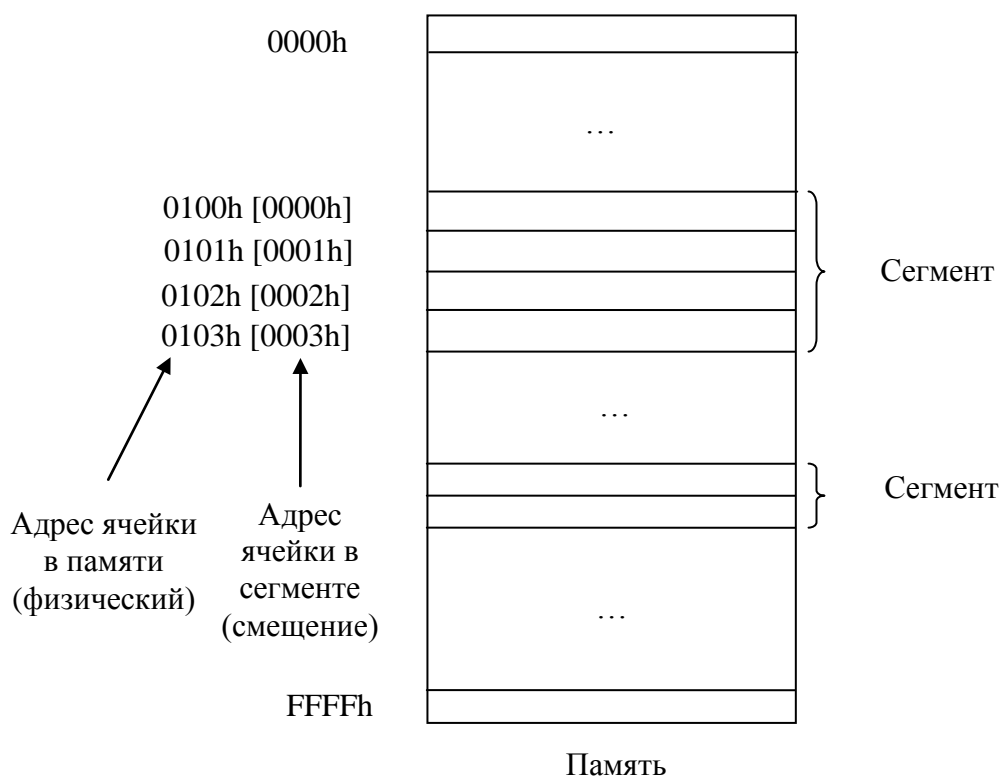
**Длина сегмента** – это количество входящих в него ячеек памяти.

Сегменты могут иметь различную длину. Все ячейки, расположенные внутри сегмента, перенумеровываются, начиная с нуля. Адресация ячеек внутри сегмента ведется относительно начала сегмента; адрес ячейки в сегменте называется **смещением** или **эффективным адресом - ЕА** (относительно начального адреса сегмента).

На рис. 1 представлены примеры сегментов, указаны адреса ячеек в памяти, входящих в сегмент, и их смещения (в квадратных скобках) относительно начала сегмента (*0100h*).

В общем случае программа, размещенная в памяти, может иметь следующие сегменты: сегмент данных для хранения операндов, сегмент кода для хранения операторов программы и сегмент стека – дополнительную память для временного размещения информации. Начальные адреса сегментов помещаются микропроцессором в соответствующие сегментные регистры, о которых пойдет речь в дальнейшем.

Чтобы вычислить физический (абсолютный) адрес ячейки в памяти, необходимо сложить начальный адрес сегмента и смещение.



**Рис.1.** Размещение сегментов в памяти

## 1.2. Регистры процессора Intel 8086

Рассмотрим регистры на примере базового процессора Intel 8086, который содержит всего 14 двухбайтовых регистра. В современных процессорах их гораздо больше и большей разрядности. Однако в качестве базовой модели, в частности, для языка ассемблера, используется 14-регистровая память процессора. Регистры процессора образуют так называемую микропроцессорную память (МПП).

МПП Intel 8086 состоит из следующих регистров:

1. *Регистры общего назначения (РОН), или универсальные:* *AX* – (*AH, AL*), *BX* – (*BH, BL*), *CX* – (*CH, CL*), *DX* – (*DH, DL*). Могут использоваться для временного хранения любых данных, при этом можно работать с каждым регистром целиком как двухбайтовым, а можно отдельно, с каждой его однобайтовой половиной.

Каждый из РОН может использоваться и как специальный при выполнении некоторых конкретных команд. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют *регистрами АЛУ*:



– *AX/AH/AL (Accumulator register)* – *аккумулятор*. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно. Например, при выполнении операций умножения и деления используется для хранения первого числа, участвующего в операции, и результата операции после ее завершения.

– *BX/BH/BL (Base register)* – *базовый* регистр. Применяется для хранения базового адреса некоторого объекта в памяти (например, массивов).

– *CX/CH/CL (Count register)* – *регистр-счетчик*. Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды. К примеру, команда организации цикла *loop* кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра *CX*.

– *DX/DH/DL (Data register)* – *регистр данных*. Так же, как и регистр *AX/AH/AL*, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно. Используется как расширение регистра – аккумулятора при работе с 32-разрядными числами.

2. *Регистры смещений: SP, BP, SI, DI*. Являются неделимыми двухбайтовыми регистрами и предназначены для хранения относительных адресов ячеек памяти внутри сегментов (смещений относительно начала сегментов).

В архитектуре процессора на программно-аппаратном уровне поддерживается такая структура данных, как *стек*. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

– *SP (Stack Pointer)* – *смещение вершины стека*. Содержит указатель вершины стека в текущем сегменте стека.

– *BP (Base Pointer)* – *смещение начального адреса поля памяти*, непосредственно отведённого под стек. Предназначен для организации произвольного доступа к данным внутри стека.

Следующие два регистра используются для поддержки цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов:

– *SI (Source Index)* – индекс источника. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике.

– *DI (Destination Index)* – индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

Большинство из перечисленных регистров могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать.

3. В программной модели микропроцессора имеется четыре сегментных регистра: *CS*, *SS*, *DS*, *ES*. Фактически в этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Текущий сегмент можно указать с помощью загрузки соответствующего сегментного регистра.

Процессор поддерживает следующие типы сегментов:

1) *Сегмент кода*. Содержит команды программы. Для доступа к этому сегменту служит регистр *CS (code segment register)* – *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ процессор.

2) *Сегмент данных*. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр *DS (data segment register)* – *сегментный регистр данных*, который хранит адрес сегмента данных текущей программы.

3) *Сегмент стека*. Этот сегмент представляет собой область памяти, называемую *стеком*. Работу со стеком процессор организует по следующему принципу: *последний записанный в эту область элемент выбирается первым*. Для доступа к этому сегменту служит регистр *SS (stack segment register)* – *сегментный регистр стека*, содержащий начальный адрес сегмента стека.

4) *Дополнительный (расширенный) сегмент данных*. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре *DS*. Если программе недостаточно одного сегмента данных, то она имеет возможность ис-

пользовать еще дополнительный сегмент данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре *DS*, при использовании дополнительного сегмента данных его адрес требуется указывать явно с помощью специальных *префиксов переопределения сегментов* в команде. Адрес дополнительного сегмента данных должен содержаться в регистре *ES* (*extension data segment registers*).

4. В процессор включены несколько регистров, которые постоянно содержат информацию о состоянии, как самого микропроцессора, так и программы, команды которой в данный момент загружены на исполнение. К этим регистрам относятся:

– *Регистр указателя команды IP* – имеет размер два байта и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра *CS* в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра *IP*.

– *Регистр флагов FLAGS* или *слово состояния процессора (PSW – Processor State Word)* имеет размер два байта и содержит одноразрядные признаки или флаги.

Всего в регистре девять флагов: шесть из них *условные* или *статусные*, отражают результаты операций, выполненных ОУ, остальные три – *управляющие*, определяют режим исполнения программы.

1. Статусные флаги. К ним относятся:

1.1. *CF (Carry Flag)* – флаг переноса. Устанавливается в 1, если при выполнении арифметических и некоторых операций сдвига возникает «перенос» из старшего разряда.

1.2. *PF (Parity Flag)* – флаг чётности. Проверяет младшие 8 битов результатов над данными. Чётное число единиц приводит к установке этого флага в 1, нечётное – в 0.

1.3. *AF (Auxiliary Carry Flag)* – флаг логического переноса в двоично-десятичной арифметике. Устанавливается в 1, если арифметическая операция приводит к переносу или займу четвёртого справа бита однобайтового операнда. Используется при арифметических операциях над двоично-десятичными кодами и кодами ASCII.

1.4. *ZF (Zero Flag)* – флаг нуля. Устанавливается в 1, если результат операции равен 0, в противном случае ZF обнуляется.

1.5. *SF (Sign Flag)* – флаг знака. Устанавливается в 1, если результат арифметической операции является отрицательным, в 0, если результат положительный.

1.6. *OF (Overflow Flag)* – флаг переполнения. Устанавливается в единицу при арифметическом переполнении, когда результат выходит за пределы разрядной сетки.

2. Управляющие флаги. К ним относятся:

2.1. *TF (Trap Flag)* – флаг трассировки. Единичное состояние этого флага переводит процессор в режим пошагового выполнения программы.

2.2. *IF (Interrupt Flag)* – флаг прерываний. При нулевом состоянии этого флага прерывания запрещены, при единичном – разрешены (о механизме прерываний речь пойдёт в следующей главе).

2.3. *DF (Direction Flag)* – флаг направления. Используется в строковых операциях для задания направления обработки данных; при единичном состоянии строки обрабатываются «справа налево», при нулевом – «слева направо».

Расположение флагов в регистре PSW показано на рис. 2. Свободные биты отведены для использования в будущем.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

**Рис. 2.** Схема расположения флагов в регистре PSW

### 1.3. Вычисление физического адреса в процессоре Intel 8086

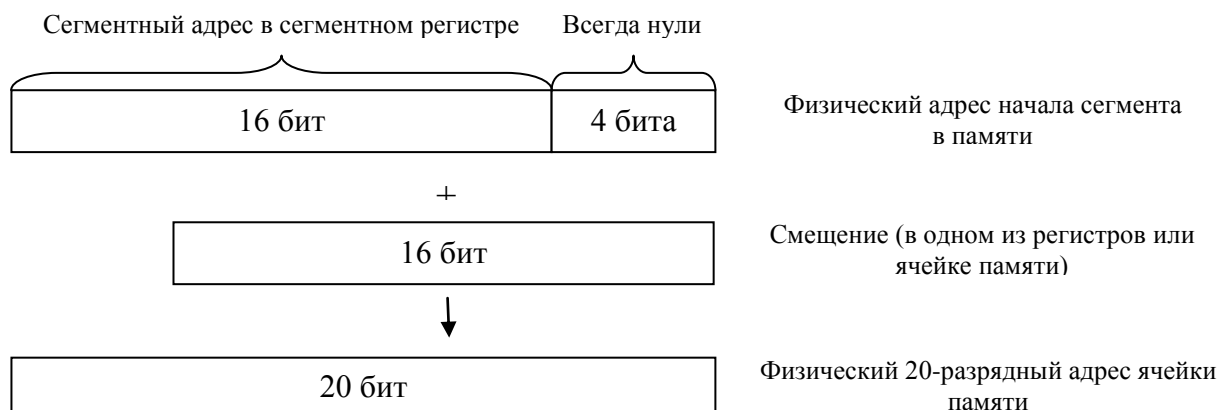
Процессор Intel 8086, на примере которого будут рассматриваться принципы функционирования процессоров, имеет 20-разрядную адресную шину, что соответствует адресному пространству  $2^{20} = 1$  Мбайт. Вследствие этого физический адрес ячейки памяти для этого процессора состоит из 20 разрядов (битов).

Для получения 20-разрядного физического адреса ячейки памяти требуется сложить начальный адрес сегмента памяти, в котором располагается эта ячейка, и смещение этой ячейки относительно начала сегмента (см. рис. 3).

Адрес начала сегмента всегда начинается с параграфа, т.е. выбирается таким образом, чтобы 20-разрядный адрес ячейки был кратен 16 (содержал четыре последних нуля).

Это позволяет сегментный адрес без четырёх младших битов (т.е., делённый на 16) хранить в одном из сегментных двухбайтовых регистров (*SS*, *DS*, *CS*, *ES*).

При вычислении физического адреса процессор умножает содержимое сегментного регистра на 16 и добавляет к полученному 20-разрядному адресу двухбайтовое смещение.



**Рис. 3.** Формирование физического адреса ячейки памяти

Система команд является одной из важнейших архитектурных характеристик процессора и ВМ в целом. Она определяет совокупность операций, реализуемых процессором. В понятие «система команд» входят: 1) форматы команд и обрабатываемых данных; 2) список команд и их функциональное назначение; 3) способы адресации данных и команд.

#### **1.4. Классификация и структура команд процессора**

По функциональному признаку все команды процессора можно разделить на следующие группы:

1. команды пересылки данных и ввода – вывода;
2. команды арифметических и поразрядных логических операций;
3. команды передачи управления.

*Команды пересылки данных* обеспечивают обмен информацией между регистрами микропроцессора, а также внешние обмены данными при передаче в процессор из памяти или устройства ввода и из процессора в память или устройство вывода. В этих командах обычно указывается направление передачи, источник и (или) приёмник данных.

Например, в языке ассемблера к командам этой группы можно отнести команду пересылки *MOV*, команду загрузки *LOAD*, команды записи в порт и чтения из порта *YBB*, *IN* и *OUT*, соответственно. Также сюда часто включают команды помещения данных в стек *PUSH* и извлечения данных из стека *POP*.

*Примеры:*

mov ax,4; Переслать в регистр ax значение 4

mov al,a; Переслать в регистр al значение по адресу a

mov a,al; Переслать в ячейку по адресу a содержимое регистра al

in al,61h; Считать в регистр al значение порта 61h

out 61h,al; Вывести в порт 61h содержимое регистра al

В число команд арифметических и поразрядных логических операций в большинстве случаев входят команды простейших арифметических операций, например, *ADD* (сложить), *SUB* (вычесть), а также логических операций, например, *AND* («И»), *OR* («ИЛИ») и т.п. К арифметическим командам относят также команды арифметических и логических сдвигов, а к командам логических операций – команда сравнения *COMPARE* (неразрушающего вычитания). В число команд этой группы могут входить команды сложных арифметических операций: умножение, деление (есть не во всех процессорах), команды обработки данных с плавающей точкой, команды мультимедийной обработки.

*Примеры:*

add ax,4; Сложить содержимое регистра ax со значением 4

sub al,a; Вычесть из регистра al значение по адресу a

mul a; Умножить значение по адресу a на содержимое регистра al

or al,00000001b; Установить в регистре al значение нулевого  
; бита в 1

and ah,0; Сбросить все биты регистра al в 0

Команды передачи управления используются для изменения последовательности выполнения команд при наличии программных ветвлений: команд условных и безусловного (*JMP*) переходов, обращении к подпрограммам (*CALL*) и выхода из них (*RETURN*). Команды условных переходов реализуют передачи управления в зависимости от значения флагов в регистре *PSW*. С их помощью процессор выполняет одну из возможных ветвей продолжения программы. Обычно в системе команд имеется несколько команд условных переходов.

*Пример:*

...

jmp m1; Перейти на команду с адресом m1

...

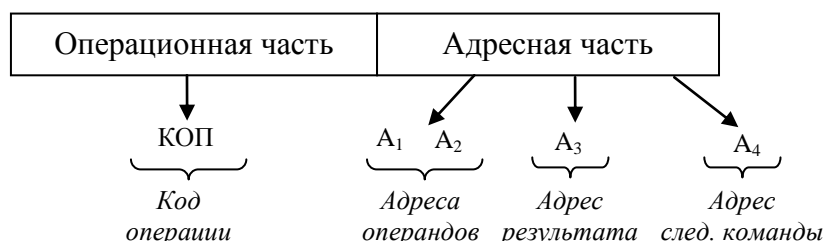
m1: mul a; Умножить значение по адресу a на содержимое  
; регистра al

В современных процессорах системы команд наряду с традиционными командами, перечисленными выше, содержат в своём составе группы команд, расширяющие функциональные возможности микропроцессора по обработке информации, управлению его работой, а также обеспечивающие реализацию многозадачного защищённого режима работы.

В системы команд конкретных процессоров могут входить команды, не вписывающиеся в предложенную классификацию. Подобные команды не отражают общих принципов построения программ и рассматриваются как дополнительные.

Выполнение команды (машинной операции) разделено на более мелкие этапы – **микрооперации** (микрокоманды), во время которых выполняются определённые элементарные действия. Конкретный состав микроопераций определяется системой команд и логической структурой ВМ. Последовательность микрокоманд, реализующих данную операцию (команду), образует **микропрограмму операции**. Интервал времени, в течение которого выполняется одна или одновременно несколько микроопераций, называется **машинным тактом**. Границы тактов задаются синхросигналами, которые вырабатываются генератором синхросигналов.

В общем случае, команда микропроцессора содержит две части: *операционную* и *адресную* (см. рис. 4).



**Рис. 4.** Прямая адресация

Соглашение о распределении разрядов между этими частями команды и способе кодирования информации определяет структуру (формат) команды.

В *операционной* части команды содержится код операции (КОП), обеспечивающий кодирование  $2^n$  операций (где  $n$  – число двоичных разрядов, отведённых под операционную часть команды) и определяющий, какие устройства в процессоре или вне его при этом будут задействованы.

В  $k$ -разрядной *адресной* части команды содержится информация об адресах операндов, участвующих в выполнении операции. В об-

шем случае адресная часть команды должна содержать четыре адресных поля  $A1$ ,  $A2$ ,  $A3$ ,  $A4$ . Они предназначены для задания адресов операндов ( $A1$ ,  $A2$ ), адреса результата ( $A3$ ) и адреса следующей команды ( $A4$ ). В качестве адресов  $A1, \dots, A3$  могут использоваться адреса ячеек оперативной памяти и адреса регистров микропроцессорной памяти, в качестве адреса  $A4$  – только адреса ячеек оперативной памяти.

При использовании полного набора адресов формат команды оказывается громоздким. Следует отметить, что не для всех операций необходим полный набор адресов  $A1 - A4$ . В зависимости от указываемого числа адресов команды подразделяются на *0-адресные* или *безадресные* (например, **NOP** – ничего не делать), *1-адресные* (например, **PUSH AX** – поместить содержимое регистра  $AX$  в вершину стека), *2-адресные* (например, **ADD AH,AL** – сложить содержимое регистров  $AH$  и  $AL$ ), *3-адресные* (тогда операция сложения, например, могла бы выглядеть следующим образом: **ADD AH,AL,BX** – сложить содержимое регистров  $AH$  и  $AL$  с сохранением результата в регистре  $BX$ ) и *4-адресные* (операция сложения могла быть записана, например, так **ADD AH,AL,BX, 0020** - сложить содержимое регистров  $AH$  и  $AL$  с сохранением результата в регистре  $BX$  и последующей загрузкой команды по смещению  $0020$  в сегменте кода).

Практически во всех микропроцессорах адрес  $A4$  исключён. Это обусловлено тем, что большинство команд относятся к линейным участкам алгоритмов, и такие команды могут быть размещены в ячейках памяти с последовательно возрастающими адресами. В этом случае для получения адреса следующей команды к начальному адресу сегмента кода достаточно добавить её смещение в сегменте кода, что удобно реализовать с помощью указателя команд ( $IP$ ). Такой способ адресации команд называется *естественным*, а реализующие его процессоры называются *процессорами с естественным способом адресации команд*. При нарушении естественного порядка следования команд (ветвлениях, циклах) используются специальные команды передачи управления, в которых содержится адрес перехода, но не используются адреса операндов. Процессоры, в адресном поле команд которых используется адрес  $A4$ , называются *процессорами с принудительным способом адресации команд*.

Использование адреса результата  $A3$  во многих случаях также оказывается избыточным. Это обосновывается тем, что результат арифметических и логических операций над двумя операндами обыч-



но может быть помещён на место одного из операндов, который в дальнейшем, скорее всего, использоваться не будет. При этом в 2-адресных командах в адресное поле необходимо вводить дополнительные разряды, показывающие, кто из них является источником, а кто – приёмником информации. В процессорах с аккумуляторной архитектурой число адресов в адресной части команды уменьшено до одного. В них один из операндов, размещённых в аккумуляторе, неявно задаётся кодом команды, и результат помещается в аккумулятор.

В безадресных командах осуществляется неявное задание операнда. К таким командам относятся команды управления процессором (например, пуска, останова и т.д.) и команды для работы со стеком (операнд, адресуемый указателем *SP*, неявно задаётся кодом команды). Безадресные команды имеют предельно сокращённый формат, но не могут самостоятельно образовать функционально полную систему команд и поэтому применяются только вместе с адресными.

Формат команд влияет на время решения задач, затраты памяти, сложность процессора и зависит от класса решаемых задач. В частности, для научно-технических расчётов, в которых большой объём занимают многошаговые вычисления, более эффективными оказываются 1-адресные команды, а при использовании стекового процессора – и безадресные команды. Для задач управления, где большую долю составляют пересылки и логические операции, эффективными являются 2-адресные команды. Исходя из сказанного выше, следует отметить, что в современных процессорах обычно используются безадресные, 1-адресные и 2-адресные команды. 3-адресные команды используются крайне редко, а 4-адресные не используются совсем.

Для процессора Intel 8086 используются следующие форматы команд:

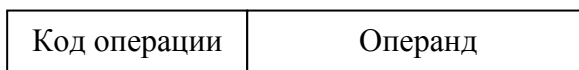
- 1) формат команды **“регистр - регистр”** (длина 2 байта);
- 2) формат команды **“регистр - память”** (длина 2 – 4 байта);
- 3) формат команды **“регистр - непосредственный операнд”** (длина 3 – 4 байта);
- 4) формат команды **“память- непосредственный операнд”** (длина 3 – 6 байтов).

В зависимости от структуры данных и их местонахождении доступ к ним может осуществляться различными способами. Рассмотрим режимы адресации данных для микропроцессора Intel 8086.

## 1.5. Способы адресации данных процессора Intel 8086

Способы адресации данных определяют механизмы вычисления эффективных адресов операндов в памяти и доступа к операндам. Выделяют следующие способы (режимы) адресации:

1. *Непосредственный*. Позволяет задавать фиксированные значения операнда непосредственно в адресной части команды, т.е., искомое значение является частью команды (см. рис. 5). Такой режим адресации удобен при работе с константами.



**Рис. 5 .** Непосредственная адресация

*Примеры:*

`mov ax, 5564h`; Переслать в регистр *ax* значение *5564*

                  ; в шестнадцатеричной системе счисления

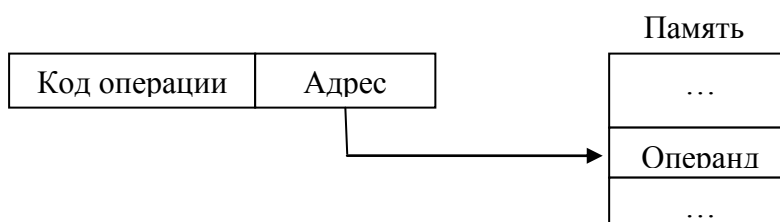
`mov ah, 'A'`; Переслать в регистр *ah* символ *A*

`add al, 11010011b`; Сложить содержимое регистра *al* с числом  
                  ; *11010011* в двоичной системе счисления

Следует помнить, что непосредственный операнд может быть задан только как операнд-источник.

Недостатком непосредственной адресации является необходимость расширения формата команд за счёт указания самого операнда в адресном поле команды.

2. *Прямой*. Адрес операнда содержится в коде команды (см. рис. 6). Используется при работе с переменными и константами, местоположение которых в памяти не меняется в процессе выполнения задачи.



**Рис. 6.** Прямая адресация

Таким образом, в коде команды указывается смещение операнда в памяти, а соответствующая метка (символьный адрес) предварительно описана в сегменте данных программы.

*Пример:*

```
d_s segment
mm dw 3154h
d_s ends
c_s segment
assume ds:d_s, cs:c_s
begin:
...
mov ax, mm; по адресу mm пересылается 3154h
...
c_s ends
end begin
```

После выполнения выделенной подчёркиванием команды в регистре *ax* будет записано значение по адресу *mm* в памяти, т.е., число *3154h*.

3. *Регистровый*. Искомое значение операнда содержится в определённом командой регистре, т.е., в адресном поле команды указывается адрес регистра (см. рис. 7).

Код операции	Регистр
--------------	---------

**Рис. 7.** Регистровая адресация

*Примеры:* `mov ax, cx`; Переслать в регистр *ax* содержимое  
; регистра *cx*  
`add ah, al`; Сложить содержимое регистров *ah* и *al*

Регистровую адресацию легко отличить от всех остальных по тому признаку, что все операнды команд являются регистрами. Такие команды являются наиболее компактными и выполняются быстрее других типов команд, поскольку отсутствуют обращения к памяти.

4. *Регистровый косвенный*. Является частным случаем косвенной адресации, когда адрес, указываемый в команде, является указателем ячейки, содержащей смещение операнда в памяти (см. рис. 8).

Фактически в команде указывается адрес адреса, причём в качестве регистра адреса может выступать базовый регистр *BP* или индексные регистры *SI* или *DI*.

Следует отметить, что в разных процессорах для размещения адреса могут использоваться и другие регистры.

Косвенная адресация является более эффективной, чем прямая, поскольку в адресном поле команды указывается только адрес регистра, который короче полного адреса операнда в памяти. Однако при

этом режиме адресации требуется предварительная загрузка регистра косвенным адресом памяти, на что расходуется дополнительное время.

*Пример:*

```
d_s segment
  mm dw 3154h
```

```
d_s ends
```

```
c_s segment
```

```
assume ds:d_s, cs:c_s
```

```
begin:
```

```
...
```

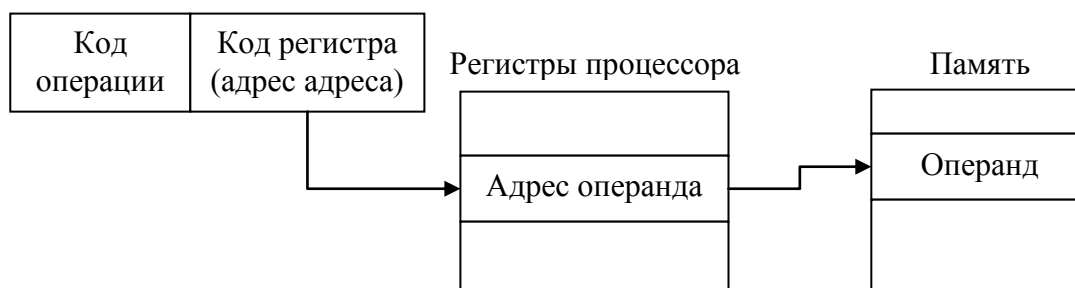
```
lea si, a;      загрузить в регистр si адрес ячейки mm
```

```
mov ax, [si];   в регистр ax пересылается значение по адресу,  
                  ; указанному в регистре si (т.е. число 3154h)
```

```
...
```

```
c_s ends
```

```
end begin
```



**Рис. 8.** Косвенная адресация

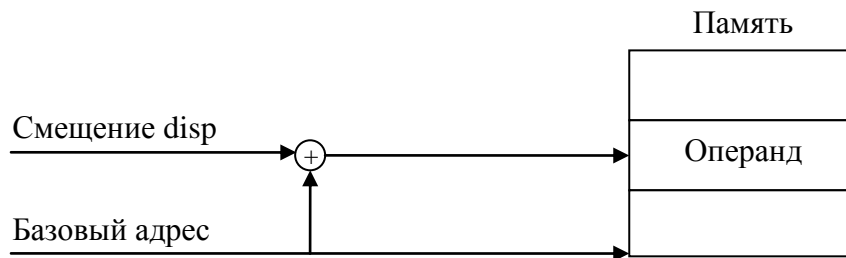
Косвенную адресацию удобно использовать при решении задач, когда оставляя неизменным адрес регистра в команде, можно изменять содержимое ячейки с этим адресом.

5. *Регистровый относительный.* Является обобщением методов адресации, обеспечивающих вычисление эффективного адреса (*EA*) или смещения операнда в памяти в виде суммы базового значения адреса и «смещения» *disp*, указываемого в команде (формула 1.1) и (рис. 9) .

$$EA = \left\{ \begin{matrix} BP \\ BX \\ SI, DI \end{matrix} \right\} + \{ \text{Смещение } disp \}. \quad (1.1)$$

Относительную адресацию широко применяют как для адресации памяти, представленной в виде блоков (например, сегментов), так и для адресации специальных структур данных: массивов, записей и др.

В зависимости от способа использования адресуемого в команде регистра различают *индексный* и *базовый* режимы адресации.



**Рис. 9.** Формирование эффективного адреса при относительной адресации

5.1. *Индексный*. Применяется для обработки упорядоченных массивов данных; при этом каждый элемент массива определяется собственным номером. Тогда базовый адрес массива задаётся смещением *disp*, указываемым в команде, а значение индекса (номер ячейки с элементом массива) определяется содержимым индексного регистра (формула 1.2).

$$EA = \{SI\}_{DI} + disp. \quad (1.2)$$

Индексная адресация удобна, если необходимо записать или считать список данных из последовательных ячеек памяти не подряд, а с некоторым шагом, указанным в индексе.

*Пример:*

d\_s segment

mas db 3,5,1,8,9,'\$'; По адресу *mas* определена последовательность  
; из 6 однобайтовых элементов  
;(с учётом символа \$)  
; \$ - признак конца последовательности

d\_s ends

c\_s segment

assume ds:d\_s, cs:c\_s

begin:

...

**mov si,3**; в *si* – смещение элемента (8) относительно начала  
; массива, т.е., адреса *mas*

**mov ah, mas[si]**; *mas*- смещение массива в сегменте данных  
; в *ah* – значение элемента массива *mas*  
; со смещением в *si*, т.е. 8

...

c\_s ends

end begin

Следует отметить, что элементы массива располагаются в памяти в смежных (соседних ячейках) относительно начального (нулевого) элемента, адрес которого в сегменте данных (смещение) является адресом начала всего массива в сегменте данных. Местоположение элемента в массиве характеризуется номером ячейки относительно начального адреса массива (начиная с нуля) и вычисляется как произведение индекса элемента на размер, занимаемый одним элементом в байтах. Чтобы получить доступ к элементу массива в сегменте данных, необходимо задать адрес начала массива и номер ячейки в массиве с нужным элементом.

В приведённом выше примере начало массива определяется адресом *mas*; требуемый элемент (8) является третьим (индексация начинается с нуля); каждый элемент массива занимает 1 байт в памяти. Чтобы определить местоположение данного элемента в сегменте данных, требуется в индексный регистр (в примере *SI*) поместить номер ячейки в массиве с нужным элементом (в примере  $3 = 1 \times 3$ ) и сложить его с начальным адресом массива (в примере *mas*).

5.2. *Базовый*. Применяется для доступа к структурам данных переменной длины. Тогда базовый адрес, определяющий начало набора элементов, хранится в базовом регистре, а смещение в команде определяет расстояние до определённого элемента (формула 1.3).

$$EA = \left\{ \begin{matrix} BP \\ BX \end{matrix} \right\} + disp. \quad (1.3)$$

Этот режим адресации удобно использовать для записей – структур данных, содержащих поля различной длины и, возможно, различных типов. В записях с полями различной длины содержимое адресуемого регистра соответствует началу записи, а смещение в команде – расстоянию в записи (номеру ячейки относительно начала записи, с которой начинается требуемое поле записи).

Рассмотрим пример организации записи о сотрудниках некоторого отдела и доступа к ней и её полям. Условимся, что все поля символьные.

*Пример:*

<i>worker</i>	<i>struc</i>	; информация о сотруднике		
<i>nam</i>	<i>db</i>	30	<i>dup</i>	(' ') ; фамилия, имя, отчество
<i>position</i>	<i>db</i>	30	<i>dup</i>	(' ') ; должность
<i>age</i>	<i>db</i>	2	<i>dup</i>	(' ') ; возраст
<i>standing</i>	<i>db</i>	2	<i>dup</i>	(' ') ; стаж
<i>salary</i>	<i>db</i>	5	<i>dup</i>	(' ') ; оклад в рублях
<i>worker</i>	<i>ends</i>			

```

d_s segment
;описание одного сотрудника
    sotr1 worker <'Иванов Пётр Сергеевич',
    'программист', '30', '8', '15000'>
d_s ends
c_s segment
    assume ds:d_s, cs:c_s
begin:
    ...
;загружаем в bx адрес начала записи (базовый адрес)
lea bx, sotr1
;в ax – значение по адресу bx+смещение по полю age
;т.е., от начала записи находим ячейки,
;содержащие информацию о возрасте
mov ax, word ptr [bx].age
    ...
c_s ends
end begin

```

Поля записи располагаются в смежных ячейках, но могут занимать разное количество байтов. В рассмотренном примере в описании структуры (записи) *worker* поля *nam* и *position* занимают по 30 байтов каждое, поля *age* и *standing* – по 2 байта каждое, а поле *salary* – 5 байтов. Имя *sotr1* в сегменте данных является адресом начала записи в сегменте данных, а имена полей фактически задают адреса (номера) ячеек, начиная с нуля, относительно начала записи. Чтобы определить местоположение конкретного поля записи (в примере *age*), необходимо знать адрес начала записи (в примере *sotr1*) и номер ячейки, относительно начала записи, с которой начинается требуемое поле (*age*).

6. *Базово-индексный*. Используется для доступа к элементам массива, адресуемого указателем. Базовый адрес массива задаётся указателем базы (базовым регистром), а номер элемента массива – содержимым индексного регистра (формула 1.4).

$$EA = \{BP\}_{BX} + \{SI\}_{DI}. \quad (1.4)$$

*Пример:*

```

d_s segment
    mas db 3,5,1,8,9,'$'; По адресу mas определена последовательность
                        ; из 6 однобайтовых элементов
                        ; (с учётом символа $)
                        ; $ - признак конца последовательности
d_s ends

```

```

c_s segment
assume ds:d_s, cs:c_s
begin:
...
    mov si,3 ; в si – смещение элемента (8) относительно начала
                ; массива, т.е., адреса mas
    lea bx,mas; в регистр bx загружается адрес начала массива mas
    mov ah, bx[si]; bx- смещение (адрес) массива в сегменте данных
                ; в ah – значение элемента массива mas
                ; со смещением в si, т.е. 8
...
c_s ends
end begin

```

В отличие от индексной адресации, где начальный адрес массива задаётся прямо в команде в виде смещения, в базово-индексном режиме начальный адрес массива предварительно загружается в один из базовых регистров (в примере *BX*).

Как и при косвенной адресации, такой режим адресации данных удобно использовать при работе со сложными структурами, когда по неизменному адресу ячеек изменяется их содержимое.

7. *Относительный базовый индексный*. Используется для адресации элементов в указываемом массиве записей. Базовый адрес массива задаётся указателем базы, номер записи (т.е., элемента массива) определяется содержимым индексного регистра, а смещение в команде указывает расстояние до записи (формула 1.5).

$$EA = \{BP\}_{BX} + \{SI\}_{DI} + disp. \quad (1.5)$$

Чтобы узнать стаж работы третьего сотрудника, сначала нужно в базовый регистр загрузить адрес начала массива (в примере *mas\_sotr*), затем определить смещение (номер ячейки) в массиве, с которого начинается запись о третьем сотруднике (в примере – это вторая запись, поскольку индексация начинается с нуля), и записать его в индексный регистр. Для определения номера нужной ячейки требуется размер одной записи (*worker*) в байтах (директива *type*) умножить на индекс записи в массиве. И, наконец, в найденной записи следует найти нужное поле (в примере *standing*).

Таким образом, чтобы получить доступ к конкретному полю массива записей, сначала необходимо определить начало массива, в нём найти нужную запись, а уже в ней – требуемое поле.



*Пример:*

```
...
d_s segment
;опишем массив из 5 сотрудников со значениями по
;умолчанию
mas_sotr worker 5 dup (<>)
d_s ends
c_s segment
assume ds:d_s, cs:c_s
begin:
...
;в bx – адрес начала массива сотрудников
lea bx, mas_sotr
;в si – смещение второй (начиная с нуля) записи
mov si, (type worker)*2
; в ax – стаж второго сотрудника
mov ax,[bx][si].standing
...
c_s ends
end begin
```

Выбор режима адресации определяется конкретной задачей и во многих случаях очевиден. Однако возникают ситуации, когда для обращения к одним и тем же элементам данных допускается использовать нескольких способов адресации. В конечном итоге, при написании программы сам пользователь осуществляет выбор конкретного режима адресации.

## **1.6. Способы адресации команд процессора Intel 8086**

Способы адресации команд определяют правила вычисления адреса команды в сегменте кода, которая должна быть выполнена следующей, при наличии в программе команд передачи управления (условных и безусловных переходов).

В зависимости от того, в каком сегменте кода находится требуемая команда и явно или нет указывается её адрес, выделяют следующие режимы адресации команд:

1. *Внутрисегментный прямой.* Команда, к которой осуществляется переход, находится в том же сегменте кода, что и текущая команда перехода, т.е. при выполнении перехода содержимое регистра *CS* не изменяется.

Эффективный адрес перехода (смещение команды в сегменте кода) вычисляется как сумма текущего содержимого указателя команд

*IP* и 8- или 16- битного относительного смещения (длины пропускаемых команд). Данный режим допустим в условных и безусловных переходах. Например,

```
...
cmp ah, al; Сравнивается содержимое регистров ah и al
jne met; Если содержимое регистров не равно,
           ; выполняется переход на команду с адресом met
inc al; Увеличение содержимого регистра al на 1
met: inc ah; Увеличение содержимого регистра ah на 1
...
```

Если содержимое регистров *ah* и *al* не равно (команда *jne*), то осуществляется переход к команде с меткой *met* путём добавления к текущему содержимому регистра *IP* длины пропускаемой команды (*inc al*).

**2. Межсегментный прямой.** Команда, к которой осуществляется переход, находится в другом сегменте кода по отношению к текущей команде перехода, т.е. при выполнении перехода изменяется содержимое регистра *CS* и регистра *IP*.

В команде указывается пара: сегмент и смещение. Начальный адрес нового сегмента кода загружается в сегментный регистр *CS*, а смещение – в регистр *IP*. Данный режим допустим только в командах безусловного перехода.

В качестве примера рассмотрим программу, состоящую из двух сегментов кода.

```
c_s1 segment
assume cs:c_s1
m1: mov ah,4
add ah,7
jmp c_s:m2 ; Прямой межсегментный переход в сегмент c_s
c_s1 ends

c_s segment
assume cs:c_s
begin:
jmp c_s1:m1; Прямой межсегментный переход в сегмент c_s1
m2: mov ah,4ch
int 21h
c_s ends
end begin
```

Сегмент кода *c\_s* является главным (в нём содержится начало и конец программы) и с него начинается выполнение программы. Вы-

деленные жирным шрифтом команды осуществляют прямой межсегментный переход. При этом адрес перехода, указанный в этих командах, является четырёхбайтовым (два старшие байта указывают начальный адрес сегмента, два младшие байта – смещение команды в этом сегменте). При выполнении первого перехода в регистр *CS* записывается начальный адрес сегмента *c\_s1*, а в регистр *IP* – адрес команды в этом сегменте кода (*m1*). Второй межсегментный переход выполняется аналогично.

Таким образом, при прямой адресации в адресном поле команды содержится адрес перехода – адрес, по которому размещается следующая выполняемая команда.

3. *Внутрисегментный косвенный*. В этом случае двухбайтовый адрес перехода размещается в ячейках памяти по некоторому адресу (смещению) в сегменте данных. В команде перехода это смещение указывается в регистре процессора или ячейке памяти с помощью любого режима адресации данных, кроме непосредственного.

Содержимое указателя команд *IP* заменяется соответствующим содержимым регистра или ячейки памяти. Данный способ допустим только в командах безусловного перехода.

Например,

d\_s segment

**adr dw met**; По адресу *adr* указан адрес команды,  
; на которую должен быть выполнен переход

d\_s ends

c\_s segment

assume ds:d\_s, cs:c\_s

begin:

mov ax, d\_s

mov ds, ax

add ah, al; Складывается содержимое регистров *ah* и *al*

**jmp adr**; Выполняется переход на команду, адрес которой  
; указан по смещению *adr* в сегменте данных

inc al; Увеличение содержимого регистра *al* на 1

**met:** inc ah; Увеличение содержимого регистра *ah* на 1

mov ah, 4ch

int 21h

c\_s ens

end begin

В приведённом выше примере адрес перехода указан в ячейке памяти по смещению *adr* относительно начала сегмента данных. В

команде безусловного перехода используется прямой режим адресации данных для указания местоположения адреса перехода, т.е. адрес адреса.

4. *Межсегментный косвенный*. В этом режиме четырёхбайтовый адрес перехода размещается в смежных ячейках памяти по некоторому адресу (смещению) в сегменте данных. В команде перехода это смещение указывается в регистре процессора или ячейке памяти с помощью любого режима адресации данных, кроме непосредственного и регистрового. Содержимое регистров *CS* и *IP* заменяется содержимым двух смежных слов памяти, хранящихся по этому смещению в сегменте данных. Младшее слово загружается в регистр *IP*, старшее – в регистр *CS*. Данный режим допустим только в командах безусловного перехода.

В качестве примера рассмотрим программу, состоящую из двух сегментов кода и одного сегмента данных. В сегменте данных по адресу *a* указан адрес перехода в виде двойного машинного слова.

```
d_s segment
a dd c_s1:m1; По адресу a указан адрес перехода в сегмент c_s1
               ; по смещению m1
d_s ends

c_s1 segment
assume cs:c_s1
m1: mov ah,4
    add ah,7
    jmp c_s:m2
c_s1 ends

c_s segment
assume ds:d_s, cs:c_s
begin:
    lea bp, a; В регистр bp загружается адрес ячейки a
    jmp dword ptr [bp]; Выполняется безусловный межсегментный
                       ; переход по адресу, указанному в регистре bp
m2: mov ah,4ch
    int 21h
c_s ends
end begin
```

Перед выполнением команды перехода в регистр *BP* загружается адрес ячейки *a*, по которому хранится адрес перехода в сегмент *c\_s1* на команду с адресом *m1*. В команде перехода с помощью регистра *BP* указывается местонахождение адреса перехода в виде двойного

машинного слова с использованием косвенной регистровой адресации данных.

Рассмотренные способы адресации команд используются практически во всех системах команд, расширяя или сокращая список команд конкретного процессора.

### 1.7. Общий формат ассемблерной команды

В самом общем виде команда на языке ассемблера выглядит следующим образом:

Метка: Мнемоника Операнд, Операнд; Комментарий

*Метка* (символьный адрес команды в сегменте кода) представляет собой идентификатор, то есть последовательность букв и цифр, начинающаяся с буквы. Символы метки могут разделяться знаком подчеркивания. Все имена регистров являются зарезервированными и их использовать в качестве метки нельзя. Они используются для указания соответствующих регистров. Символьный адрес команды является необязательным. Он указывается в команде тогда, когда на неё ссылаются в командах условного или безусловного переходов.

*Команда (мнемоника)* указывает транслятору с ассемблера, какое действие должен выполнить данный оператор.

*Операнды* – регистры, метки (адреса) данных или непосредственные данные. В зависимости от формата команды количество операндов может варьироваться: ни одного (безадресная команда), один (одноадресная команда) или два (двухадресная команда). Разрядность операндов должна совпадать.

*Комментарий* служит для пояснения действий команды или директивы ассемблера. После точки с запятой комментарий записывается на одной строке. Для продолжения комментария на последующих строках он записывается после точки с запятой. Комментарий является необязательным.

Таким образом, из всех компонентов команды в обязательном порядке указывается сам оператор и необходимые для его выполнения операнды.

*Примеры:*

```
add ah, al; Сложить содержимое регистров ah и al  
met: inc ah; Увеличение содержимого регистра ah на 1  
mov ah,4
```

## 1.8. Определение данных

В сегменте данных можно зарезервировать определённое количество байтов для размещения исходных данных или результатов работы программы, а также указать их начальные значения. Формат операторов резервирования и инициализации данных в общем случае имеет вид:

Метка Мнемоника Операнд,...,Операнд ; Комментарий

*Метка* (символьный адрес) обозначает смещение (номер ячейки в сегменте данных) первого резервируемого байта. К метке в сегменте данных обладает теми же свойствами, что и метка в сегменте кода, но является обязательной.

*Мнемоника* определяет длину каждого операнда:

1) **DB** (определить байт). Диапазон для целых чисел без знака: 0...255, для целых чисел со знаком: -128...127.

2) **DW** (определить слово – два байта). Диапазон для целых чисел без знака: 0...65535, для целых чисел со знаком: -32768...32767.

3) **DD** (определить двойное слово – четыре байта).

Диапазон для целых чисел без знака: 0...4294967295, для целых чисел со знаком: -2147483648...2147483647.

*Операнды* показывают инициализируемые данные или объем резервируемого пространства. Выражение может содержать константу или символ ? для неопределенного значения.

*Примеры:*

```
Data_byte DB 104
Data_word DW 100H,FFH,-5
Data_DW DD 5*25,0FFFDH,1
Data_str DB 'H','E','L','L','O'
Data_str1 DB 'HELLO'
```

При определении большого числа ячеек можно применять оператор повторения

DUP (Операнд,...,Операнд).

*Примеры:*

Arr DB 30 DUP(1,2) – зарезервирует по адресу Arr 30 однобайтовых ячеек с начальными значениями 1 в нечетных и 2 в четных байтах;

MM1 DB 15 DUP(14) – означает, что по адресу MM1 находятся 15 байт содержащих шестнадцатиричную цифру 0EH (14 в десятичной системе).

**Arr\_DW DW 4 DUP(?)** – определяет по адресу *Arr\_DW* 4 ячейки, содержащих произвольную информацию.

Операнды могут задаваться в различных системах счисления: для двоичной системы счисления после значения операнда ставится символ *B*, для шестнадцатеричной – символ *H*, для десятичной – ничего.

*Примеры:*

**Data\_word DW 100H** – по адресу *Data\_word* зарезервировано 2 байта с первоначальным значением *100* в шестнадцатеричной системе счисления (или *256* в десятичной).

**Met DB 01010011B** – по адресу *Met* зарезервирована однобайтовая ячейка с первоначальным значением *01010011* в двоичной системе счисления (или *83* в десятичной).

**ARG DB 15** – по адресу *ARG* зарезервирована однобайтовая ячейка с первоначальным значением *15* в десятичной системе счисления.

Далее рассмотрим основные команды языка ассемблера для процессора Intel 8086, необходимые для выполнения заданий лабораторного практикума.

## 1.9. Основные команды языка ассемблера

Прежде чем перейти к рассмотрению основных команд языка ассемблера, следует ознакомиться с условными обозначениями, используемыми в дальнейшем. Перечень сокращений и условных обозначений приведен в табл. 1.1.

Таблица 1.1. *Перечень сокращений и условных обозначений*

Сокращение	Смысловое значение
1	2
OPR	Операнд
SRC	Операнд- источник
DST	Операнд- получатель
REG	Регистр
RSRC	Регистр- источник
RDST	Регистр- получатель
CNT	Счетчик
DISP	Смещение
ADDR	Адрес

1	2
EA	Эффективный адрес
SEG	Сегмент
DATA	Непосредственный операнд

### 1.9.1. Команды передачи данных

Предназначены для пересылок данных, адресов и непосредственных операндов в регистры или в ячейки памяти. Их описание представлено в табл. 1.2.

Таблица 1.2. Формат команд передачи данных

Название команды	Мнемоника и формат команды	Описание действия
Переслать	MOV DST, SRC	(DST) ← (SRC) Копирование содержимого операнда – источника в операнд – приёмник
Загрузить эффективный адрес	LEA DST, SRC	(REG) ← (SRC) Загрузка эффективного адреса (смещения) операнда – источника в регистр процессора
Обменять	XCHG OPR1, OPR2	(OPR1) ← (OPR2) (OPR2) ← (OPR1) Обмен значениями между операндами

При выполнении указанных выше команд ни один из флажков не изменяется.

Что касается режимов адресации, то операнд – приёмник не может быть непосредственным значением и не может быть сегментным регистром *CS*.

В команде *LEA* операнд *REG* не может быть сегментным регистром, а источник не может иметь непосредственный или регистровый режим.

В команде *MOV* один из операндов должен быть регистром.

В команде *XCHG* хотя бы один из операндов должен быть регистром, но ни один из операндов не может быть сегментным регистром.



*Примеры:*

mov ah,8; Переслать в регистр *ah* десятичное значение 8  
mov bx,mm; Переслать в регистр *bx* значение по адресу *mm*  
; в сегменте данных  
mov ah,al; Переслать в регистр *ah* содержимое регистра *al*  
mov mm,bx; Переслать в ячейку по адресу *mm* в сегменте данных  
; содержимое регистра *bx*  
lea bp,mm; Загрузить смещение ячейки *mm* в регистр *bp*  
xchg bx,mm; Обменять содержимое регистра *bx* со значением  
; по адресу *mm* в сегменте данных  
xchg ch,ah; Обменять содержимое регистров *ch* и *ah*  
xchg mm,ax; Обменять значение по адресу *mm* в сегменте  
; данных с содержимым регистра *ax*

### **1.9.2. Команды сложения и вычитания двоичных чисел**

**Целое двоичное число** – это формат представления числовой информации в ВМ в двоичном коде. Для описания таких чисел в программе используются директивы описания данных *DB*, *DW* и *DD*, рассмотренные в пункте 1.8 практикума.

Команды сложения и вычитания применяются для выполнения арифметических операций над целыми двоичными числами. Описание команд представлено в табл. 1.3.

Таблица 1.3. *Формат команд сложения и вычитания*

Название команды	Мнемоника и формат команды	Описание действия
Сложить	ADD DST, SRC	$(DST) \leftarrow (SRC) + (DST)$ Сложение двух целочисленных операндов
Сложить с переносом	ADC DST, SRC	$(DST) \leftarrow (SRC) + (DST) + (CF)$ Сложение двух целочисленных операндов с учётом значения флага переноса <i>CF</i>
Вычесть	SUB DST, SRC	$(DST) \leftarrow (DST) - (SRC)$ Вычитание целочисленных операндов
Вычесть с заемом	SBB DST, SRC	$(DST) \leftarrow (DST) - (SRC + CF)$ Вычитание двух целочисленных операндов с учётом значения флага переноса <i>CF</i>

При выполнении указанных выше команд сложения и вычитания модифицируются все флажки условий.

Особенностью данных команд является то, что они записывают результат на место первого операнда (приёмника), вследствие чего операнд – приёмник должен находиться в регистре процессора. Операнд – источник может иметь любой режим адресации.

*Примеры:*

add ah,8; Сложить содержимое регистра *ah* с десятичной 8

adc ax,mm; Сложить содержимое регистра ax со значением

; по адресу *mm* с учётом флага переноса *cf*

sub ah,1bh; Вычесть из содержимого регистра *ah*

; шестнадцатеричное значение *1b*

sbb bh,bl; Вычесть из содержимого регистра *bh* содержимое *bl*

; с учётом флага переноса *cf*

### 1.9.3. Однооперандные команды двоичной арифметики

Применяются для увеличения или уменьшения на единицу операнда и для изменения знака операнда. Их описание представлено в табл. 1.4.

Таблица 1.4. *Формат однооперандных команд арифметики*

Название команды	Мнемоника и формат команды	Описание действия
Инкремент	INC OPR	$(\text{OPR}) \leftarrow (\text{OPR}) + 1$ Увеличение операнда на 1
Декремент	DEC OPR	$(\text{OPR}) \leftarrow (\text{OPR}) - 1$ Уменьшение операнда на 1
Изменить знак	NEG OPR	$(\text{OPR}) \leftarrow 0 - (\text{OPR})$ Вычисление двоичного дополнения операнда

При выполнении рассмотренных выше команд модифицируются все флажки условий, но команды *INC* и *DEC* не воздействуют на флажок *CF*.

Относительно режимов адресации, в командах *INC*, *DEC*, *NEG* не допускается непосредственный режим.

Отрицательные числа в ВМ представляются в дополнительном коде.

**Двоичное дополнение (дополнительный код)** некоторого отрицательного числа – это результат инвертирования каждого бита двоичного числа, равного по модулю исходного отрицательного числа, плюс единица.

Например, рассмотрим десятичное число -3. Модуль данного числа в двоичном представлении равен *00000011*. Инвертируем все

биты исходного числа и получаем *11111100*. Затем к полученному значению добавляем единицу, что в результате даёт *11111101*. Именно так число *-3* представляется в компьютере.

Чтобы не преобразовывать исходное число в двоичное представление, вычислить дополнительный код можно и таким способом: от максимального числа (зависит от формата модуля отрицательного числа) отнимается модуль отрицательного числа и к полученному результату добавляется 1.

Для примера возьмём снова число *-3*. Его модуль – число *3* – по формату соответствует байту. Тогда от максимального однобайтового значения *255* отнимем *3* и добавим к полученному значению *1*. В результате дополнительный код числа *-3* равен *253*, что соответствует двоичному его двоичному представлению, представленному выше (*11111101*).

*Примеры:*

*inc al*; Увеличить содержимое регистра *al* на *1*

*dec a*; Уменьшить значение по адресу *a* на *1*

*neg cx*; Изменить знак содержимого регистра *cx*

#### **1.9.4. Команды умножения и деления двоичных чисел**

Применяются для выполнения операций умножения и деления над двоичными числами. Их описание представлено в табл. 1.5.

Таблица 4.5. *Формат команд умножения и деления*

Название команды	Мнемоника и формат команды	Описание действия
Умножить без знака	MUL SRC	См. табл. 1.6
Умножить со знаком	IMUL SRC	См. табл. 1.6
Делить без знака	DIV SRC	См. табл. 1.7
Делить со знаком	IDIV SRC	См. табл. 1.7

Особенностью операций умножения и деления является наличие в команде всего одного операнда (сомножителя или делителя), который может быть задан с помощью любого режима адресации данных, кроме непосредственного (находиться в памяти или в регистре). Вторым операнд задан неявно; его местоположение фиксировано и зависит от размера операндов. Знаки результатов в операциях со знаком определяются по алгебраическим правилам.

Варианты размеров сомножителей, мест размещения второго операнда и результата для операции умножения представлены в табл. 1.6, а для операции деления – в табл. 1.7.

Если результат по размеру совпадает с размером сомножителей, то флаги *CF* и *OF* после завершения операции раны нулю, в противном случае - устанавливаются в единицу. Это значит, что результат вышел за пределы младшей части произведения и состоит из двух частей, что необходимо учитывать при дальнейшей работе. Остальные флаги не определены.

Таблица 1.6. *Расположение операндов при умножении*

Первый сомножитель	Второй сомножитель	Результат
Байт	AL	16 битов в AX; AL – младшая часть результата, AH – старшая часть результата $(AX) \leftarrow (SRC) * (AL)$
Слово	AX	32 бита в паре DX:AX; AX – младшая часть результата, DX – старшая часть результата $(DX:AX) \leftarrow (SRC) * (AX)$

Как следует из табл. 1.6, при умножении однобайтовых значений первый сомножитель указывается в самой команде, второй сомножитель должен быть помещён в регистр AL, а произведение после выполнения команды умножения помещается в регистр AX. Процесс умножения двухбайтовых сомножителей аналогичен, за исключением того, что второй сомножитель должен быть помещён в регистр AX, а произведение командой умножения сохраняется по частям в двух регистрах: DX и AX.

*Пример:*

mov al,5; Переслать в регистр al десятичное значение 5  
 mov bh,4; Переслать в регистр bh десятичное значение 4  
 mul bh; Умножить содержимое регистра al на bh  
 mov cx,ax; Переписать значение произведения из регистра ax  
           ; в регистр cx

Таблица 1.7. *Расположение операндов при делении*

Делимое	Делитель	Частное	Остаток
Слово (16 бит) в регистре AX	Байт	Байт в регистре AL $(AL) \leftarrow (AX) / (SRC)$	Байт в регистре AH $(AH) \leftarrow (AX) / (SRC)$
Двойное слово (32 бита), в DX – старшая часть в AX – младшая часть	Слово	Слово в регистре AX $(AX) \leftarrow (DX:AX) / (SRC)$	Слово в регистре DX $(DX) \leftarrow (DX:AX) / (SRC)$

Делитель может находиться в регистре или в памяти и иметь размер 8, или 16 битов. Местонахождение делимого фиксировано: если

делитель имеет размер 1 байт, то делимое должно помещаться в регистр *AX*; если делитель является двухбайтовым, то старшие два байта делимого должны размещаться в регистре *DX*, а младшие два байта – в регистре *AX*. Результатом команды деления являются частное и остаток от деления, которые также размещаются в строго определённых регистрах (см. табл. 1.7).

После выполнения операции деления содержимое флагов не определено, но возможно возникновение исключения с номером ноль (так называемое «деление на ноль») в случаях, когда делитель равен нулю или частное не входит в отведенную для него разрядную сетку.

*Пример:*

```
mov ax,20; Переслать в регистр ax десятичное значение 20  
mov bh,4; Переслать в регистр bh десятичное значение 4  
neg bh; Изменить знак числа, хранящегося в регистре bh  
idiv bh; Разделить с учётом знака содержимое регистра ax на bh  
mov cl,al; Переписать остаток из регистра al в регистр cl  
mov ch,ah ; Переписать остаток из регистра ah в регистр ch
```

### **1.9.5. Логические команды**

Логические команды выполняют логические операции над битами операндов. Размерность операндов должна быть одинакова.

Логические команды наиболее часто используются для селективных (выборочных) установок, инвертирования, сброса или проверки битов в операнде – получателе в соответствии с двоичным значением операнда – источника. Такие действия часто встречаются в операциях над битами регистров и данных ввода-вывода. При этом операнд источник называют *маской*, а сама операция называется *маскированием*. Описание логических команд представлено в табл. 1.8.

Команда логического сравнения *TEST* выполняет операцию логического умножения над операндами; при этом сами операнды не изменяются. Результат операции формируется во временной памяти: бит результата равен 1, если соответствующие биты операндов равны 1, в противном случае бит результата равен 0.

Команда *NOT* не воздействует на флажки. Остальные команды сбрасывают *OF* и *CF*, оставляют *AF* не определённым и устанавливают *CF*, *ZF*, *PF* по обычным правилам.

Касательно режимов адресации, в команде *NOT* не допускается непосредственный операнд. В остальных командах один из операндов должен быть регистром. Другой операнд может иметь любой режим адресации.

Таблица 1.8. Формат логических команд

Название команды	Мнемоника и формат команды	Описание действия
Инвертировать	NOT OPR	$(OPR) \leftarrow \text{not } OPR$ Инвертирование всех битов операнда
Объединить по «ИЛИ»	OR DST, SRC	$(DST) \leftarrow (DST) \text{ or } (SRC)$ Выполнение операции логического «ИЛИ» над соответствующими парами битов источника и приёмника
Объединить по «И»	AND DST, SRC	$(DST) \leftarrow (DST) \text{ and } (SRC)$ Выполнение операции логического «И» над соответствующими парами битов источника и приёмника
Сложить по MOD2 («исключающее ИЛИ»)	XOR DST, SRC	$(DST) \leftarrow (DST) \text{ xor } (SRC)$ Выполнение операции логического «Исключающее ИЛИ» над соответствующими парами битов источника и приёмника
Проверить	TEST OPR1, OPR2	$OPR1 \text{ and } OPR2$ Выполнение операции логического сравнения над соответствующими парами битов двух операндов

*Примеры:*

mov al, 01001101b; Переслать в регистр *a*/ двоичное  
; значение *01001101b*

not al; Инвертировать биты содержимое регистра *a*/  
or al, 10000001b; Установить нулевой и седьмой биты  
; в регистре *a*/ в *1*

and al, 11110101b; Сбросить первый и третий биты в содержимом  
; регистра *a*/

test al, 00000001b; Проверить, является ли содержимое регистра *a*/  
; нечётным числом (нулевой бит должен быть 1)

jnz m1; Выполнить переход на команду с адресом *m1*, если нулевой  
; бит содержимого регистра *a* равен 1 (число нечётное)  
; В этом случае результат команды *test* равен 1 и значение  
; флага *zf* равно 0

### **1.9.6. Команды сдвигов и циклических сдвигов**

Эти команды также обеспечивают манипуляции над отдельными битами, перемещая биты операнда влево или вправо на определенное число битов, в зависимости от кода операции. Количество битов, на которое выполняется сдвиг, определяется счетчиком сдвигов *CNT*. Значение счетчика может задаваться статически (непосредственно во втором операнде) или динамически (в регистре *CL* перед выполнением команды сдвига). Исходя из размерности регистра *CL*, очевидно, что значение счётчика сдвигов *CNT* может находиться в диапазоне от 0 до 255. Однако в целях оптимизации процессор воспринимает только значения пяти младших счётчика, что сокращает границы его значений до диапазона от 0 до 31.

Все команды сдвига воздействуют на флаг переноса *CF*. По мере сдвига битов за пределы операнда они сначала попадают во флаг переноса *CF*. В командах сдвига влево с правой стороны операнда «вдвигаются» нули, а старшие биты «выдвигаются» с левой стороны и теряются, но последний из них сохраняется во флаге *CF*. Команды сдвига вправо аналогичным образом сдвигают биты вправо.

Описание команд линейных и циклических сдвигов представлено в табл. 1.9.

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они воспринимают сдвигаемые значения как числа со знаком и особым образом работают со знаковым битом (седьмым) числа. Но арифметический сдвиг вправо не помещает слева нули, а дублирует в старшие биты знак операнда.

Команды арифметического сдвига позволяют выполнить «быстрое» умножение и деление операнда на степени двойки. Например, сдвиг числа влево на один разряд аналогичен его умножению на 2, а сдвиг числа вправо на один разряд аналогичен делению его на 2. Преимущество этих команд по сравнению с традиционными командами умножения и деления заключается в скорости исполнения: команды сдвига выполняются быстрее.

Команды циклического сдвига отличаются от команд сдвига тем, что операнд считается «кольцом», в котором выдвигаемые с одной стороны биты вдвигаются с другой стороны.

В командах простого циклического сдвига сдвигаемый бит одновременно и вдвигается в операнд с другого конца, и становится значением флага переноса *CF*.

Таблица 1.9. Формат команд сдвигов и циклических сдвигов

Название команды	Мнемоника и формат команды	Описание действия
Сдвинуть логически влево	SHL OPR,CNT	
Сдвинуть арифметически влево	SAL OPR,CNT	Не сохраняет знака, но устанавливает флаг <i>OF</i> в случае смены знака очередным выдвигаемым битом. В остальном полностью аналогична команде <i>SHL</i> .
Сдвинуть логически вправо	SHR OPR,CNT	
Сдвинуть арифметически вправо	SAR OPR,CNT	Сохраняет знак, восстанавливая его после сдвига каждого очередного бита. В остальном аналогична команде <i>SHR</i> .
Сдвинуть циклически влево	ROL OPR,CNT	
Сдвинуть циклически вправо	ROR OPR,CNT	
Сдвинуть циклически влево через перенос	RCL OPR,CNT	
Сдвинуть циклически вправо через перенос	RCR OPR,CNT	



В командах циклического сдвига через перенос сдвигаемый бит сначала помещается во флаг переноса *CF*. Только при следующем выполнении той же команды находящийся во флаге *CF* бит вталкивается с другой стороны операнда, а во флаг *CF* помещается следующий бит сдвигаемого числа.

Флажки *SF*, *ZF*, *PF* модифицируются командами линейного сдвига, но команды циклических сдвигов на них не воздействуют. Флажок *OF* имеет смысл, если только счетчик сдвига равен 1.

Команды сдвигов влияют на состояние флажка *AF*, но оно определенного смысла не имеет.

Что касается режимов адресации, то *OPR* может иметь любой режим адресации, кроме непосредственного режима.

*Примеры:*

```
mov al,01001101b; Переслать в регистр a/ двоичное
                  ;значение 01001101b
shl al; 2; Сдвинуть содержимое регистра a/ влево на 2 бита
          ; В регистре a/ появится значение 00110100b
          ; Во флаге cf появится значение шестого бита, т.е. 1
mov al,10001101b; Переслать в регистр a/ двоичное
                  ;значение 10001101b (выделенный бит – знак числа)
sar al, 3; Сдвинуть содержимое регистра a/ арифметически влево
          ; на 3 бита
          ; В регистре a/ появится значение 11110001b
          ; Во флаге cf появится значение 1
mov al,01001101b; Переслать в регистр a/ двоичное
                  ;значение 01001101b
rol al; 2; Сдвинуть содержимое регистра a/ циклически влево
          ; на 2 бита
          ; В регистре a/ появится значение 00110101b
          ; Во флаге cf появится значение шестого бита, т.е. 1
mov al,00000011b; Переслать в регистр a/ двоичное
                  ;значение 01001101b
rcl al; 1; Сдвинуть содержимое регистра a/ циклически влево
          ; с переносом на 1 бит
          ; В регистре a/ появится значение 00000001b
          ; Во флаге cf появится значение нулевого бита, т.е. 1
rcr al; 1; Циклический сдвиг с переносом вправо на 1 бит
          ; В регистре a/ появится значение 10000000b
          ; Во флаге cf появится значение нулевого бита, т.е. 1
```

### 1.9.7. Команды передачи управления

Эти команды нарушают естественный порядок выполнения команд программы.

Изменение потока управления происходит при наличии команд переходов (условного и безусловного), вызова процедур.

1. *Команды перехода.* При выполнении команд перехода в счётчик команд IP принудительно записывается новое значение – новый адрес в памяти, начиная с которого будут выполняться команды.

1.1. Команда *безусловного перехода* обеспечивает переход по заданному адресу без проверки каких-либо условий.

Формат команды:

**JMP** Модификатор адрес\_перехода

Этой командой задаются внутрисегментные и межсегментные переходы.

*Модификатор* указывает вид перехода (внутрисегментный прямой, внутрисегментный косвенный, межсегментный прямой, межсегментный косвенный) и принцип изменения содержимого регистров CS и IP. Модификатор не всегда указывается в команде *JMP*.

Описание команды безусловного перехода для внутрисегментных переходов представлено в табл. 1.10.

Таблица 4.10. *Формат команды для внутрисегментных переходов*

Вариант внутрисегментного перехода	Мнемоника и формат команды	Описание действия
Прямой короткий переход (расстояние от команды JMP до адреса перехода не превышает -128 или 127 байт)	JMP OPR JMP SHORT PTR OPR	$(IP) \leftarrow (IP) + 8\text{-битное смещение, определяемое OPR}$
Прямой переход (на расстояние от 128 байт до 64 Кбайт)	JMP OPR JMP NEAR PTR OPR	$(IP) \leftarrow (IP) + 16\text{-битное смещение, определяемое OPR}$
Косвенный переход (в команде указывается не сам адрес перехода, а место, где он находится)	JMP OPR JMP WORD PTR OPR	$(IP) \leftarrow (EA)$ , где EA- эффективный адрес перехода, определяемый OPR

Прямой короткий внутрисегментный переход применяется, когда расстояние от команды *JMP* до адреса перехода находится в диапазоне от -128 байт (адрес перехода расположен до команды *JMP* в программе) до +127 байт (адрес перехода расположен после команды *JMP* в программе) байт. В последнем случае для указания короткого перехода в команде *JMP* используется модификатор *SHORT PTR*. При выполнении короткого перехода длина команды безусловного перехода составляет два байта.

Прямой внутрисегментный переход отличается от предыдущего варианта перехода тем, что расстояние между адресом перехода и командой *JMP* находится в диапазоне от 128 байт до 64 Кбайт, т.е. переходы между командами могут осуществляться в пределах всего сегмента кода. Для уточнения вида перехода может использоваться модификатор *NEAR PTR*. При выполнении внутрисегментного прямого перехода длина команды безусловного перехода составляет три байта.

В команде косвенного внутрисегментного перехода указывается не сам адрес перехода, а его местоположение, т.е. смещение (эффективный адрес) в сегменте данных. Если адрес ячейки памяти, где хранится адрес перехода, задаётся транслятору через регистр (с помощью команды *LEA*), то в команде перехода необходимо использовать модификатор *WORD PTR*, для дополнительного сообщения о том, что переход является внутрисегментным.

#### *Примеры:*

```
...
jmp short ptr met; Выполняется переход на команду с адресом met
...                ; (расстояние до 127 байт)
met: inc ah; Увеличение содержимого регистра ah на 1
...

...
met: inc ah; Увеличение содержимого регистра ah на 1
...

...
jmp met; Выполняется переход на команду с адресом met
...      ; (расстояние до -128 байт)

...
jmp met; Выполняется переход на команду с адресом met
...      ; (расстояние от 128 байт до 64 Кбайт)
met: inc ah; Увеличение содержимого регистра ah на 1
...
```

```

d_s segment
a dw met; По адресу a указан адрес команды,
           ; на которую должен быть выполнен переход
d_s ends

c_s segment
assume ds:d_s, cs:c_s
begin:
mov ax, d_s
mov ds, ax
jmp a; Выполняется ближний переход на команду, адрес которой
      ; указан по смещению a в сегменте данных
...
met: inc ah; Увеличение содержимого регистра ah на 1
...
c_s ens
end begin

d_s segment
a dw met; По адресу a указан адрес команды,
           ; на которую должен быть выполнен переход
d_s ends

c_s segment
assume cs:c_s
begin:
mov ax, d_s
mov ds, ax
lea bp,a; Загружается адрес ячейки a в регистр bp
jmp word ptr [bp]; Выполняется ближний косвенный переход
                  ; на команду, адрес которой указан в регистре bp
...
met: inc ah; Увеличение содержимого регистра ah на 1
...
c_s ens
end begin

```

Формат команды безусловного перехода для межсегментных переходов описан в табл. 1.11.

При выполнении прямого межсегментного перехода в команде указывается адрес перехода длиной четыре байта, из которых два старшие байта – начальный адрес нового сегмента кода, а младшие два байта – адрес команды (смещение) в этом сегмента кода. Команда прямого межсегментного перехода имеет длину пять байтов. Использование модификатора *FAR PTR* обязательно.

Таблица 1.11. *Формат команды для межсегментных переходов*

Вариант межсегментного перехода	Мнемоника и формат команды	Описание действия
Прямой переход	JMP FAR PTR OPR	(CS) ← начальный адрес сегмента, определяемого OPR (IP) ← смещение в сегменте из OPR
Косвенный переход	JMP FAR PTR OPR JMP DWORD PTR OPR	(IP) ← (EA), где EA- эффективный адрес, определяемый OPR (CS) ← (EA + 2), где EA- эффективный адрес из OPR

В команде косвенного межсегментного перехода в качестве операнда указывается адрес области памяти, в которой содержатся смещение в новом сегменте и начальный адрес нового всего (всего четыре байта). Если адрес области памяти в команде перехода указан прямо, то используется модификатор *FAR PTR*, если адрес области памяти указан косвенно через регистр, то используется модификатор *DWORD PTR*.

*Пример:*

d\_s segment

a dd c\_s1:m1; По адресу a указан адрес перехода в сегмент c\_s1  
; по смещению m1

d\_s ends

c\_s1 segment

assume cs:c\_s1

m1: mov ah,4

add ah,7

jmp c\_s:m2

c\_s1 ends

c\_s segment

assume ds:d\_s, cs:c\_s

begin:

lea bp, a; В регистр bp загружается адрес ячейки a

jmp dword ptr [bp]; Выполняется безусловный межсегментный  
; переход по адресу, указанному в регистре bp

m2: mov ah,4ch

int 21h

c\_s ends

end begin

Межсегментные передачи управления реализуются только командами безусловных переходов.

Не модифицируются все флажки условий.

Что касается режимов адресации, то во внутрисегментных прямых переходах и межсегментных прямых переходах используется прямой режим. Во внутрисегментных косвенных переходах не допускается непосредственный режим, а в межсегментных косвенных переходах должна адресоваться область памяти.

1.2. *Ветвление* (условный переход) происходит только при соблюдении определённого условия, в противном случае выполняется следующая по порядку команда программы. Условием, на основании которого осуществляется переход, чаще всего выступают признаки результата выполнения предшествующей арифметической или логической команды (без флага *AF*). Каждый из признаков фиксируется в своём разряде регистра флагов *PSW*. Возможен и такой подход, когда решение о переходе принимается в зависимости от состояния одного из регистров общего назначения, куда предварительно помещается результат операции сравнения *CMP*.

Формат команды *CMP* представлен в табл. 1.12.

Таблица 1.12. *Формат команды сравнения*

Название команды	Мнемоника и формат команды	Описание действия
Сравнить	<i>CMP</i> <i>OPR1</i> , <i>OPR2</i>	( <i>OPR1</i> ) - ( <i>OPR2</i> ) Выполняется сравнение путём вычитания операндов, при этом сами операнды не изменяются

Команда *CMP* устанавливает статусные флаги в зависимости от результата сравнения операндов, не меняя самих операндов.

Команды условного перехода позволяют выполнять только короткие переходы (внутрисегментные прямые переходы в диапазоне от -128 байт до +127 байт).

Команды условного перехода и их формат представлены в табл. 1.13.

Многие команды условного перехода, представленные в табл. 1.13 эквивалентны, так как в них анализируются одинаковые флаги.

Таблица 1.13. *Формат команд условного перехода*

Название команды	Мнемоника и формат команды	Критерий условного перехода (в CMP)	Значение флагов для перехода
1	2	3	4
Перейти, если равно	JE OPR	$OPR1 = OPR2$	$ZF = 1$
Перейти, если не равно	JNE OPR	$OPR1 \neq OPR2$	$ZF = 0$
Перейти, если ниже (меньше)/ не выше или равно (без знака)	JB/ JNAE OPR	$OPR1 < OPR2$	$CF = 1$
Перейти, если не ниже (меньше)/ выше или равно (без знака)	JNB/ JAE OPR	$OPR1 \geq OPR2$	$CF = 0$
Перейти, если ниже или равно/ не выше (без знака)	JBE/ JNA OPR	$OPR1 \leq OPR2$	$CF = 1$ или $ZF = 1$
Перейти, если не ниже или равно/ выше (больше) (без знака)	JNBE/ JA OPR	$OPR1 > OPR2$	$CF = 0$ и $ZF = 0$
Перейти, если меньше/ не больше (со знаком)	JL/ JNGE OPR	$OPR1 < OPR2$	$SF \neq OF$
Перейти, если не меньше/ больше или равно (со знаком)	JNL/ JGE OPR	$OPR1 \geq OPR2$	$SF = OF$
Перейти, если меньше или равно/ не больше (со знаком)	JLE/ JNG OPR	$OPR1 \leq OPR2$	$SF \neq OF$ или $ZF = 1$
Перейти, если не меньше или равно/ больше (со знаком)	JNLE/ JG OPR	$OPR1 > OPR2$	$SF = OF$ и $ZF = 0$
Перейти, если ноль	JZ OPR	$[OPR1 = OPR2]$	$ZF = 1$
Перейти, если не ноль	JNZ OPR	$[OPR1 \neq OPR2]$	$ZF = 0$
Перейти, если знак установлен	JS OPR	$[OPR1 < OPR2]$	$SF = 1$
Перейти, если знак сброшен	JNS OPR	$[OPR1 > OPR2]$	$SF = 0$
Перейти, если есть переполнение	JO OPR	-	$OF = 1$
Перейти, если нет переполнения	JNO OPR	-	$OF = 0$

Продолжение таблицы 1.13

1	2	3	4
Перейти, если паритет установлен	JP OPR	-	PF = 1
Перейти, если паритет сброшен	JNP OPR	-	PF = 0
Перейти, если перенос установлен	JC OPR	-	CF = 1
Перейти, если перенос сброшен	JNC	-	CF = 0

Рассмотрим примеры использования команд условного перехода:

```
...
sub ah,al
jz m1
add ah,3
jmp m2
m1: add al,2
m2: mov ah,4
...
```

```
...
cmp ah,al
je m1
add ah,3
jmp m2
m1: add al,2
m2: mov ah,4
...
```

Левый фрагмент иллюстрирует проверку содержимого регистров *ah* и *al* на равенство. При этом используются флаги, в частности, флаг нуля *ZF*. Предварительно выполняется вычитание содержимого регистров: если их значения равны, то в результате образуется ноль и изменяется значение флага *ZF*. Команда *jz* проверяет условие: если флаг *ZF* равен 1, то выполняется переход на команду с адресом *m1*, иначе выполняется команда сложения, следующая за командой условного перехода. Команда с адресом *m2* выполняется в любом случае. Правый фрагмент выполняет ту же проверку, но с использованием команды сравнения *cmp* и команды перехода по равенству содержимого регистров *ah* и *al* (*je m1*).

2. *Организация циклов.* Циклы организуются для многократного повторения одной или нескольких команд программы или процедуры. Цикл можно организовать, используя команды условного и безусловного переходов, рассмотренные выше, а можно с помощью специальных команд.

Формат этих команд представлен в таблице 1.14.

Команда *LOOP* и её расширения позволяет организовывать циклы, подобные циклам *for* в языках высокого уровня с автоматическим уменьшением счётчика цикла. Количество повторений содержится в



регистре *CX*, который в командах управления циклами выполняет функции счётчика цикла.

Таблица 1.14. *Формат команд циклов*

Название команды	Мнемоника и формат команды	Проверяемое условие
Зациклить	LOOP OPR	$CX \neq 0$
Зациклить, пока ноль или равно	LOOPZ/ LOOPE OPR	$CX \neq 0$ или $ZF = 0$
Зациклить, пока не ноль или не равно	LOOPNZ/ LOOPNE OPR	$CX \neq 0$ или $ZF = 1$
Переход по CX	JCXZ OPR	$CX = 0$

Команды *LOOPZ/ LOOPE* и *LOOPNZ/ LOOPNE* по принципу своей работы являются взаимнообратными. Они расширяют действие команды *LOOP* тем, что дополнительно анализируют флаг нуля *ZF*. Это даёт возможность организовывать досрочный выход из цикла по значению флага *ZF*. Обычно эти команды применяются в операциях поиска определённого значения в последовательности или при сравнении двух чисел.

Команда *JCXZ* по своему формату идентична командам условного перехода. Она выполняет проверку регистра *CX* и осуществляет переход на команду с указанным адресом, если содержимое регистра *CX* равно 0. Но, как и в командах цикла, здесь регистр *CX* тоже играет роль счётчика цикла.

За исключением команды *JCXZ*, которая не изменяет счётчик цикла *CX*, остальные команды управления циклами уменьшают текущее содержимое счётчика *CX* на 1. Затем, если проверяемое условие удовлетворяется, то выполняется переход по адресу, указанному в команде цикла. В противном случае осуществляется переход на команду, следующую после *LOOP* или её расширений, т.е. выполняется выход из цикла.

Операнд *OPR* должен быть меткой, которая находится в диапазоне от -128 до 127 байт от команды, следующей за командой цикла. Это означает, что команды цикла позволяют выполнять короткие внутри-сегментные переходы. Для работы с длинными циклами следует использовать команды условного перехода и команду *JMP*.

Часто возникает необходимость в организации вложенных циклов. Основная проблема, которая при этом возникает – как сохранить значения счётчиков *CX* для каждого цикла. Для временного хранения

значений счётчиков внешнего и внутреннего циклов можно использовать другие регистры процессора, ячейки памяти или стек.

*Примеры:*

```
...
m1: cmp ah,al; Сравнение содержимого регистров ah и al
jg m2; Если содержимое ah больше значения в al,
    ; то выполняется переход на метку m2 (выход из цикла)
add ah,3; иначе, увеличивается содержимое регистра ah на 3
jmp m1; Выполняется переход на метку m2 (начало цикла)
m2: mov ah,4
...
...
mov cx,20; Количество повторений внешнего цикла
c1: push cx ; Помещение счётчика внешнего цикла в стек
    ; Команды внешнего цикла
mov cx,10; Количество повторений внутреннего цикла
c2:
    ; Команды внутреннего цикла
loop c2; Конец внутреннего цикла
    ; Команды внешнего цикла
pop cx; Восстановление из стека счётчика внешнего цикла
loop c1; Конец внешнего цикла
...
```

В первом примере цикл организуется с помощью команд условного и безусловного переходов. Увеличение содержимого регистра *ah* на 3 будет выполняться, пока содержимое регистра *ah* меньше содержимого регистра *al*.

Во втором примере описаны вложенные циклы. Для временного хранения счётчика внешнего цикла используется стек. Перед запуском внутреннего цикла текущее содержимое регистра *cx* запоминается в стеке. Затем в *cx* помещается значение счётчика внутреннего цикла, и выполняются его команды. По окончании внутреннего цикла выполняются команды внешнего цикла. После этого из стека извлекается значение счётчика внешнего цикла для последующей проверки условий внешнего цикла.

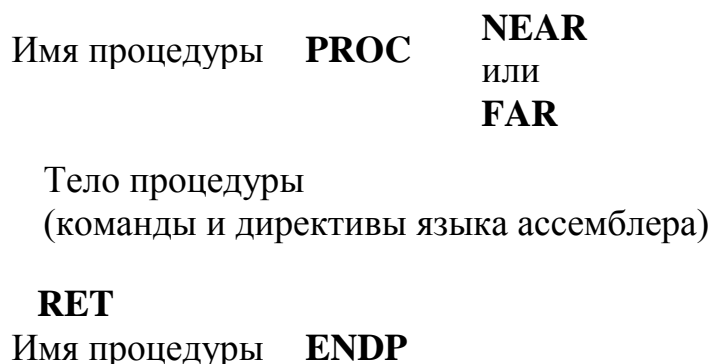
**3. Процедуры. Процедура (подпрограмма)** – это группа команд для решения конкретной подзадачи, обладающая средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку.

Другими словами, это правильным образом оформленная совокупность команд, которая, будучи однократно описана, при необхо-

димости может быть вызвана в любом месте программы. Но, в отличие от команд перехода, после выполнения процедуры управление возвращается к команде, следующей за командой вызова процедуры.

Общее описание процедуры представлено на рис. 9.

Процедура ограничивается операторами *PROC* и *ENDP*, перед которыми указывается имя процедуры. После *PROC* указывается тип процедуры: процедура ближнего вызова (директива *NEAR*) или процедура дальнего вызова (директива *FAR*). В первом случае процедура располагается в том же сегменте кода, что и основная программа, и при вызове такой процедуры выполняется внутрисегментный переход. Во втором случае процедура располагается в другом сегменте кода, и при её вызове выполняется межсегментный переход.



**Рис. 9.** Общее описание процедуры

Между этими операторами располагается тело процедуры, состоящее из команд и директив языка ассемблера. Последней командой процедуры является команда *RET*, по которой осуществляется возврат из данной процедуры в вызвавшую её программу или другую процедуру на команду, следующую за командой последнего вызова процедуры.

Вызов процедуры осуществляется командой *CALL*, за которой следует имя процедуры. Формат команды *CALL*:

**CALL** Модификатор имя\_процедуры

Модификатор может принимать значения *NEAR* или *FAR*, для обращения к процедурам ближнего или дальнего вызовов, соответственно.

Для работы с процедурами используется стек (дополнительная память, организованная в виде очереди), в который команда вызова помещает текущее значение счётчика команд (*IP*) при внутрисегментных переходах (или значения регистров *IP* и *CS* при межсег-

ментных переходах) – адрес точки возврата. При выходе из процедуры старые значения соответствующих регистров восстанавливаются из стека.

Процедурный механизм базируется на командах вызова процедуры, обеспечивающих переход из текущей точки программы к начальной команде процедуры, и командах возврата из процедуры для возврата в точку (на команду), непосредственно расположенную за командой вызова.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. В этом случае процессор воспринимает процедуру как часть исполняемого потока команд и начинает их выполнять. Процедура может размещаться:

1) *В начале программы* (до первой исполняемой команды).

```
c_s segment
assume cs:c_s
pr1 proc near
...
ret
p1 endp
begin:      ;начало программы
...
end begin
```

2) *В конце программы* (после команды корректного завершения работы и возвращения управления операционной системе – ОС).

```
c_s segment
assume cs:c_s
begin:
...
mov ah, 4ch
int 21h    ;корректное завершение работы и передача управления ОС
p1 proc near
...
ret
p1 endp
c_s ends
end begin
```

3) *Внутри тела программы или другой процедуры* (должен быть предусмотрен обход процедуры с помощью оператора *JMP*).

```
c_s segment
assume cs:c_s
begin:
...
jmp m1
p1 proc near
...
ret
p1 endp
m1: ...
mov ah, 4ch
int 21h ;корректное завершение работы и передача управления ОС
c_s ends
end begin
```

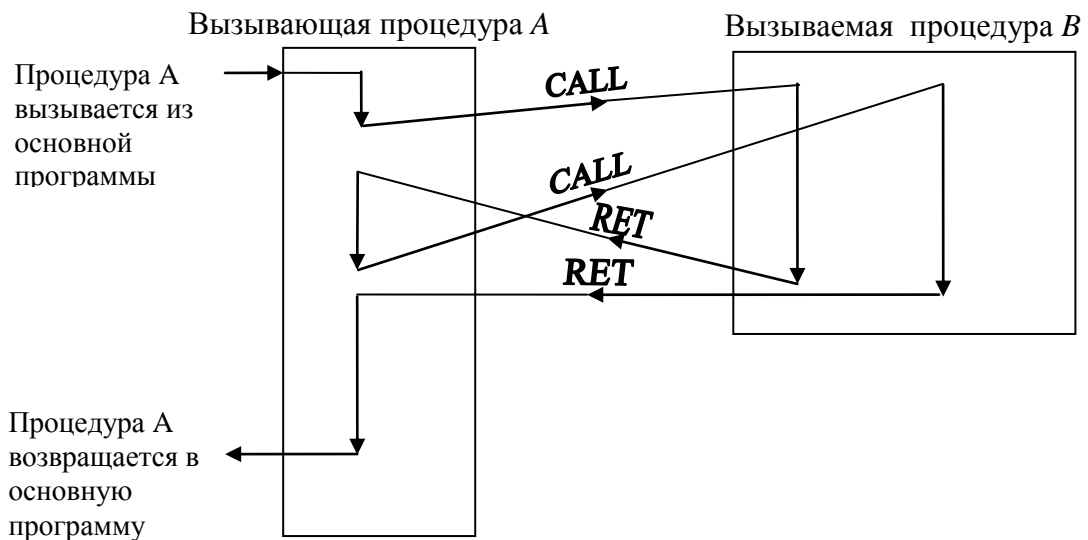
4) *В другом модуле* – часто используемы процедуры выносятся в отдельный файл, который оформляется как обычный файл ассемблера, а затем подвергается трансляции для получения объектного кода. Впоследствии этот объектный файл с помощью компоновщика можно объединить с файлом, в котором все или некоторые данные процедуры используются.

Особый интерес представляет **рекурсивная процедура** – процедура, которая вызывает сама себя либо непосредственно, либо через цепочку других процедур. Для таких процедур также используется стек, в котором помимо адреса возврата сохраняются параметры и локальные переменные для каждого вызова.

Обычно, если в программе используются процедуры, то в ней описывается сегмент стека для резервирования ячеек под дополнительную память. Если сегмент стека в программе отсутствует, то в этом случае операционная система формирует стек самостоятельно.

В обычной последовательности вызовов существует чёткое различие между вызываемой и вызывающей процедурами. Вызываемая процедура каждый раз начинается сначала, сколько бы раз не происходило обращение к ней. Для выхода из вызываемой процедуры используется команда возврата *RET*.

Взаимодействие вызывающей и вызываемой процедур иллюстрирует рис. 10.



**Рис. 10.** Взаимодействие вызывающей и вызываемой процедур

Рассмотрим в качестве примера программу, использующую вызов процедуры.

```

s_s segment stack "stack"
dw 12 dup(?)
s_s ends
d_s segment
aa dw 10
d_s ends
c_s segment
assume ss:s_s,ds:d_s,cs:c_s
begin:
mov ax,d_s
mov ds,ax
call pr1 ;вызов подпрограммы
mov ah,4ch
int 21h
pr1 proc near ;начало подпрограммы (ближний вызов)
push ax ;записать в стек содержимое регистра AX
mov ax, aa
pop ax ;выбрать из стека содержимое регистра AX
ret ;команда возврата на следующую команду после
;вызова процедуры
pr1 endp ;конец подпрограммы
c_s ends
end begin
    
```

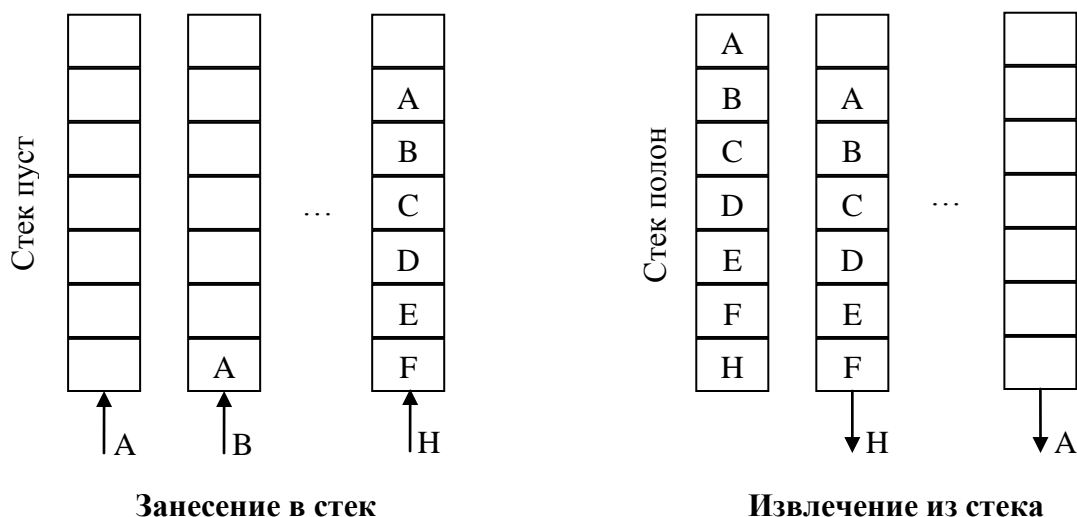
Поскольку процедура расположена в том же сегменте кода, что и основная программа и описана как процедура ближнего вызова (директива *NEAR*), то переход будет внутрисегментным

При выполнении вызова процедуры *pr1* (команда *call pr1*) в стек помещается адрес возврата – значение счётчика команд *IP*, содержащего на данный момент адрес команды, которая должна будет выполняться после текущей (*mov ah,4ch*). Значение регистра *IP* замещается новым значением – адресом первой команды процедуры. При достижении команды возврата из процедуры (*ret*) из стека в регистр *IP* записывается старое значение, что обеспечивает возврат в основную программу на команду, которая непосредственно следует за командой вызова процедуры.

### 1.9.8. Стековые команды

#### Организация стека

**Стек** – это однонаправленная очередь, данные в которую помещаются и извлекаются в строго определённом порядке. Стековая память обеспечивает такой режим работы, когда информация записывается и считывается по принципу «последним записан – первым считан» (*LIFO – Last Input First Output*). Такая память используется для временного хранения данных, например, для запоминания и восстановления регистров процессора (контекста) при обработке подпрограмм и прерываний. Работу стековой памяти поясняет рис. 11.



**Рис.11.** Логика работы стековой памяти

Когда слово *A* заносится в стек, то располагается в первой свободной ячейке. Каждое следующее записываемое слово перемещает

всё содержимое стека на одну ячейку вверх и занимает освободившуюся ячейку. Запись очередного слова после *N* приводит к переполнению стека, поскольку он рассчитан на 7 слов, и потере кода *A*.

Считывание информации из стека осуществляется в обратном порядке, т.е., начиная с кода *N*, который был записан последним. Доступ к произвольному коду в стеке формально недопустим до извлечения всех данных, записанных позже.

Занесение информации в стек называется **включением**, считывание информации из стека – **извлечением**.

В настоящее время наиболее распространённым является внешний, или аппаратно-программный, стек, в котором для хранения информации отводится область оперативной памяти. Обычно под стек отводится участок памяти с наибольшими адресами, а расширяется стек в сторону уменьшения адресов.

На рис. 12 показана схема организации стека для процессора Intel 8086. Под стек выделяется отдельный сегмент – сегмент стека, начальный адрес которого помещается в соответствующий сегментный регистр – *SS*. Адресация стека обеспечивается специальным регистром – указателем стека *SP*, в который предварительно помещается наибольший адрес области основной памяти, отведённой под стек (дно стека). Адрес последнего включённого в стек элемента называется **вершиной стека** (*TOS – Top Of Stack*). Размер стека зависит от режима работы процессора и ограничивается значением 64 Кбайт в обычном режиме (или 4 Гбайт в защищенном режиме).

В каждый момент времени доступен только один стек, начальный адрес сегмента которого содержится в регистре *SS*. Для перехода к другому стеку необходимо загрузить в *SS* его адрес. По мере записи данных в стек он растёт в сторону младших адресов памяти.

Для работы со стеком существуют две основные операции: **добавление элемента в вершину стека** (*PUSH*) и **извлечение элемента из вершины стека** (*POP*). Формат команд представлен в табл. 1.15.

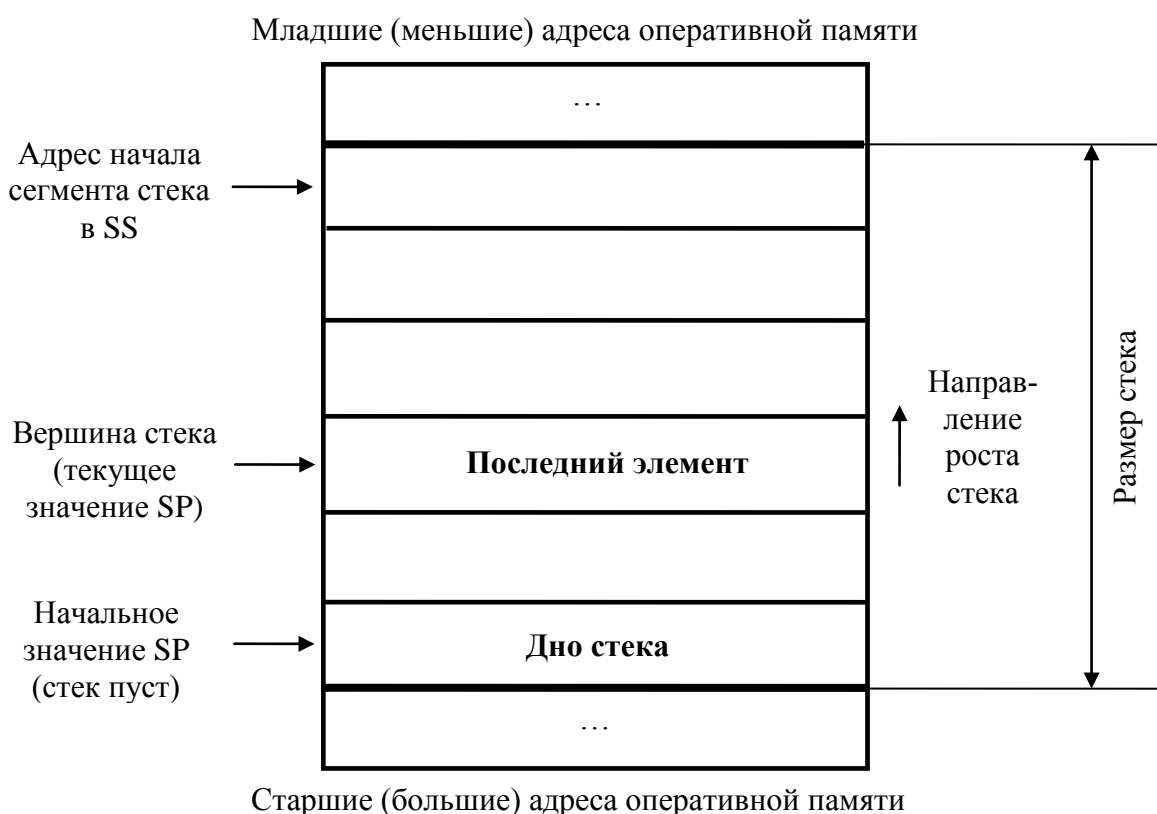
Таблица 4.14 – Формат стековых команд

Название команды	Мнемоника и формат команды	Описание действия
Включить в стек	PUSH SRC	$(SP) \leftarrow (SP) - 2$ $(SS:SP) \leftarrow (SRC)$
Извлечь из стека	POP DST	$(DST) \leftarrow (SS:SP)$ $(SP) \leftarrow (SP) + 2$



Команда *PUSH* имеет один операнд, который может быть двухбайтовым регистром, кроме регистра *CS*, или ячейкой памяти такого же размера. При записи в стек данного сначала производится уменьшение на 2 содержимого указателя стека *SP* (стек оперирует словами), а затем по адресу, указываемому парой *SS:SP*, производится запись операнда – источника.

Команда *POP* также имеет один операнд, который может быть двухбайтовым регистром, кроме регистра *CS*, или ячейкой памяти. При считывании слова из стека в качестве адреса берётся текущее содержимое указателя стека в сегменте стека (*SS:SP*), а после извлечения данного слова в операнд – приёмник содержимое *SP* увеличивается на 2.



**Рис. 12.** Схема организации стека для процессора Intel 8086

Когда стек пуст, то значение регистра *SP* равно адресу последнего байта сегмента (самому старшему адресу ячейки памяти в сегменте), выделенного под стек. Когда стек заполнен, то значение регистра *SP* становится равным значению регистра *SS*, и дальнейшее добавление элементов невозможно.

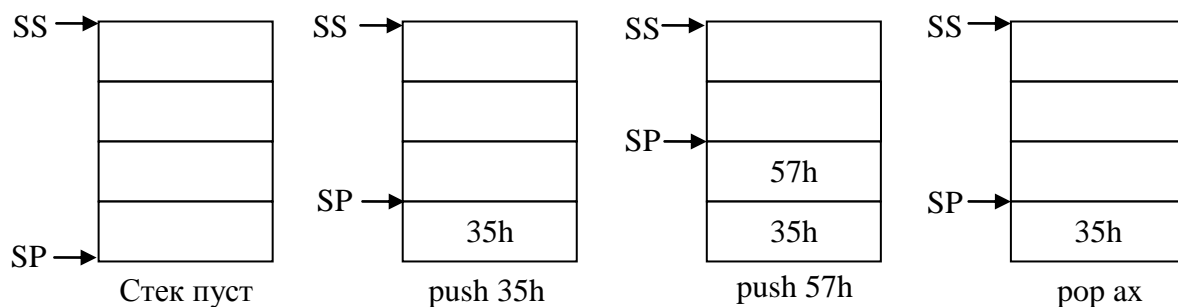
При помещении элементов в стек адрес вершины стека (содержимое регистра *SP*) уменьшается (смещается в сторону меньших ад-

ресов), а при извлечении элементов из стека – увеличивается (смещается в сторону больших адресов).

Рассмотрим следующий пример:

```
push 35h  
push 57h  
pop ax
```

Состояние стека при выполнении указанных выше команд иллюстрирует рис. 13.



**Рис. 13.** Изменение состояния стека

Изначально стек пуст, и регистр *SP* указывает на ячейку за дном стека. При включении первого значения содержимое *SP* уменьшается на 2 и затем по полученному адресу помещается 35h. Аналогично со вторым значением. При извлечении слова из стека в регистр *AX* помещается значение из вершины стека, т.е., 57h, а содержимое *SP* увеличивается на 2.

Для доступа к элементам не в вершине, а внутри стека используется регистр *BP* – указатель базы кадра стека. Например, при входе в процедуру выполняется передача нужных параметров путем записи их в стек. Если процедура также использует стек, то доступ к этим параметрам становится проблематичным. Выход заключается в том, чтобы после записи параметров в регистр *BP* записать адрес вершины стека *SP*. Значение регистра *SP* в дальнейшем будет изменяться, однако в регистре *BP* хранится адрес, используя который, можно получить доступ к переданным параметрам.

### **Использование стека для передачи параметров в процедуру**

Стек может использоваться для временного хранения значений регистров и ячеек памяти, адресов возврата из процедур, для передачи параметров в процедуры.

При передаче управления процедуре процессор автоматически записывает в вершину стека два (для процедур ближнего вызова) или четыре (для процедур дальнего вызова) байта – *адрес возврата* в вызывающую программу. Если предварительно в стек были записаны переданные процедуре параметры или указатели на них, то они окажутся под адресом возврата.

Ранее уже упоминалось, что для работы со стеком в процессоре предусмотрены три регистра *SS*, *SP* и *BP*. Процессор автоматически работает с регистрами *SS* и *SP* в предположении, что они всегда указывают на вершину стека. По этой причине их содержимое изменять не рекомендуется.

Для произвольного доступа к данным в стеке используется регистр *BP*. Для корректной работы с использованием этого регистра содержимое стека должно быть правильно проинициализировано, что предполагает формирование в нём адреса, который бы непосредственно указывал на переданные данные. Для этого в начале процедуры необходимо включить дополнительный фрагмент кода – ***пролог процедуры***. Конец процедуры также должен быть оформлен особым образом для обеспечения корректного возврата из процедуры. Фрагмент кода, выполняющий эти действия, называется ***эпилогом процедуры***. При этом нужно откорректировать содержимое стека, убрав из него ставшие ненужными аргументы, переданные и использованные в процедуре.

Например, можно использовать последовательность из *n* команд вида **ROP регистр**. Лучше всего это сделать в вызывающей программе сразу после возврата управления из процедуры.

Рассмотрим пример программы, в которой осуществляется вызов процедуры с передачей параметров в неё через стек.

Код пролога состоит из двух команд: первая команда сохраняет содержимое регистра *BP* в стеке, чтобы исключить затирание находящегося в нём значения в вызываемой процедуре; вторая команда настраивает регистр *BP* на вершину стека для осуществления прямого доступа к содержимому стека.

Для доступа к последнему аргументу достаточно сместиться от содержимого *BP* на 4 (2 первых байта занимает адрес возврата, 2 следующих – искомое значение), к предпоследнему аргументу – на 6 и так далее (для процедур ближнего вызова).

Код эпилога процедуры восстанавливает состояние программы до момента входа в процедуру.

```

s_s segment stack "stack"
    dw 12 dup(?)
s_s ends
d_s segment
    aa dw 10
d_s ends
c_s segment
    assume ss:s_s,ds:d_s,cs:c_s
begin:
    mov ax,d_s
    mov ds,ax
    push aa ;запись в стек аргумента
    call pr1 ;вызов процедуры
    pop ax ;очищаем стек, забирая аргумент в регистр ax
    .mov ah, 4ch
    int 21h ;завершение работы программы

pr1 proc near
    ;начало пролога
    push bp
    mov bp,sp
    ;конец пролога
    mov ax,[bp+4] ;доступ к аргументу по адресу aa для процедуры
    ;в регистре ax будет значение 10
    add ax,158h
    mov dx,ax
    ;начало эпилога
    mov sp,bp ;восстановление значения регистра sp
    pop bp ;восстановление значения старого bp
    ;до входа в процедуру
    ret ;возврат в вызывающую подпрограмму
    ;конец эпилога
pr1 endp

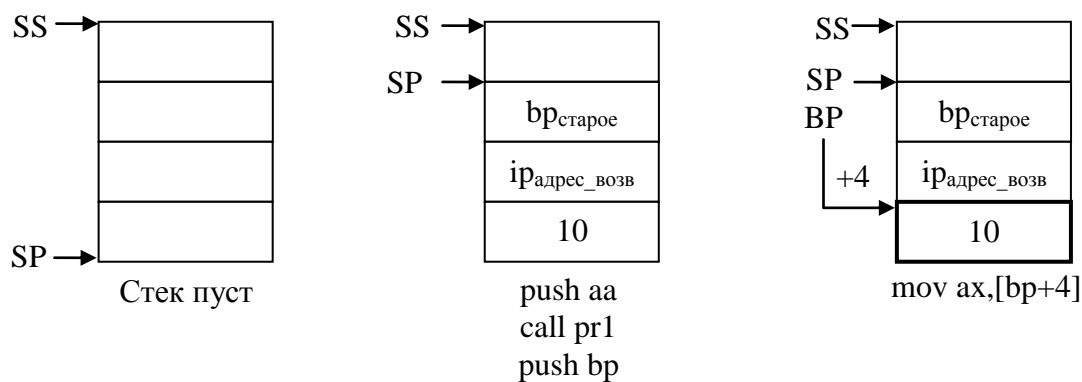
c_s ends
end begin

```

Изменение состояния стека к моменту доступа к параметрам, передаваемым в процедуру, представлено на рис. 14. В соответствии с приведённой программой, сначала в стек помещается параметр (значение по адресу *aa*), затем адрес возврата (при вызове процедуры) и, наконец, содержимое регистра *bp*. Тогда *sp* указывает на вершину стека, где хранится исходное значение регистра *bp*. После этого содержимое регистров *sp* и *bp* становится одинаковым (*mov bp,sp*), и

они оба указывают на вершину стека. Таким образом, параметр процедуры располагается ниже вершины стека на 4 байта под адресом возврата и старым значением *bp*, каждый из которых занимает по два байта. Для доступа к ячейке стека с требуемым параметром от ячейки, на которую указывает *bp*, опуститься ниже на 4 байта, пропустив старое значение *bp* и *адрес возврата*, т.е. к текущему содержимому регистра *bp* прибавить 4 байта.

Что касается способов передачи параметров в процедуру через стек, то можно передавать либо сами данные (**передача параметров по значению**), либо их адреса (**передача параметров по ссылке**). В приведённой выше программе использовался способ передачи аргументов по значению.



**Рис. 14.** Изменение состояние стека к моменту доступа к параметрам процедуры

При передаче параметров через стек по значению на их размер накладываются ограничения, связанные с размерностью стека. Кроме того, в этом случае в вызываемой процедуре обрабатываются копии параметров. Таким образом, в рассмотренном примере значение по адресу *aa* в сегменте данных не изменится, то есть останется равным 10, независимо от выполняемых над этим значением действий в процедуре.

При передаче аргументов по ссылке в вызываемой процедуре обрабатывается не копия, а оригинал передаваемых данных. Поэтому при изменении данных в вызываемой процедуре они автоматически изменяются и в вызывающей программе, поскольку изменения касаются одной области памяти.

## 2. ПРОГРАММИРОВАНИЕ УСТРОЙСТВ НА ЯЗЫКЕ АССЕМБЛЕРА

### 2.1. Прерывания, исключения и механизм их обработки

*Исключения и прерывания* обеспечивают принудительную передачу управления специальной процедуре – **обработчику системных прерываний**, который выполняет определённые действия. После завершения действий обработчик прерывания возвращает управление прерванной программе. Она должна продолжить выполнение прерванного процесса в том же самом состоянии, в котором находилась в момент прерывания. Обработчик прерываний отличается от обычной процедуры тем, что вместо связывания с конкретной программой, он размещается в фиксированной области памяти.

**Исключение** – это особый тип вызова процедуры, который происходит при определённом условии – важном, но редко встречающемся. Наиболее распространенными условиями, которые могут вызвать исключения, являются переполнение и исчезновение значащих разрядов при операциях с плавающей точкой, а также переполнение при операциях с целыми числами, нарушение защиты, неопределяемый код операции, переполнение стека, попытка запустить несуществующее УВВ, попытка вызвать слово из ячейки с нечётным адресом, деление на ноль. Если результат находится в пределах допустимого, исключение не возникает. Важно отметить, что этот вид прерывания вызывается каким-то исключительным условием, вызванным самой программой и обнаруженным аппаратным обеспечением или микропрограммой. Исключения, в свою очередь, подразделяются на *сбои, ловушки и аварийные завершения*.

**Сбой (отказ)** – это исключение, сообщение о котором выдаётся на границе команды, предшествующей команде, вызвавшей это исключение. После сообщения о сбое состояние ВМ восстанавливается в ситуацию, когда можно выполнить рестарт команды. Примером сбоя может служить обращение к несуществующему УВВ.

**Ловушка** – это исключение, сообщение о котором выдаётся на границе команды, непосредственно расположенной после команды, для которой было обнаружено данное исключение. Например, переполнение при обработке целых чисел.

**Аварийное завершение** – это исключение, не позволяющее ни точно определить команду, вызвавшую его, ни перезапустить программу, в которой произошло данное исключение. Аварийные завер-

шения используются для сообщения о серьёзных ошибках: аппаратных ошибках, противоречивых или недопустимых значениях в системных таблицах.

**Прерывания** - это изменения в потоке управления, вызванные не самой программой, а какими-то асинхронными событиями, и связанные обычно с процессом ввода – вывода. Прерывания дают возможность осуществлять операции ввода – вывода независимо от процессора. Поскольку быстродействие микропроцессора выше, чем УВВ, то процессор имеет возможность выполнять другие программы или осуществлять другие функции вместо постоянного контроля состояния присоединённых к нему периферийных устройств. Когда УВВ требует обслуживания со стороны процессора, оно сообщает ему об этом путём формирования соответствующего запроса (сигнала), по которому процессором может быть прервано выполнение текущей программы. Управление прерываниями от УВВ осуществляется контроллером прерываний, который подключён к процессору и структурно входит в его состав.

Различие между исключениями и прерываниями заключается в том, что исключения синхронны по отношению к программе, а прерывания асинхронны. Исключения вызываются программой непосредственно, а прерывания – опосредованно. Если многократно перезапускать программу с одними и теми же входными данными, то исключения будут возникать всякий раз в одних и тех же местах программы, а прерывания – нет.

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы, процессор может иметь систему защиты от прерываний: отключение системы прерываний, запрет или маскирование отдельных сигналов прерываний. Программное управление этими средствами позволяет операционной системе регулировать обработку сигналов прерываний. Процессор может либо обрабатывать прерывания сразу по их приходу, либо откладывать их обработку на некоторое время, либо полностью игнорировать. Например, если установлен в единицу флажок трассировки *TF*, то процессор выполняет одну команду программы, а затем генерирует прерывание типа 1, т.е., программа выполняется по шагам. Если сброшен флаг прерываний *IF*, то процессор не реагирует на внешние прерывания (за исключением немаскируемых). Для маскирования отдельных прерываний используется регистр масок. Управляется командами *CLI* (запретить прерывания) и *STI* (разрешить прерывания).

С каждым отдельным типом прерывания или исключением связан идентифицирующий его номер в диапазоне от 0 до 255 и соответствующий обработчик. Исключениям и немаскируемым прерываниям присвоены номера из интервала от 0 до 31, а маскируемым прерываниям – от 32 до 255. Не все из этих значений используются процессорами в настоящее время; неназначенные номера зарезервированы для использования в будущем. Номера прерываний, исключений и адреса (векторы) соответствующих обработчиков хранятся в специальной таблице – **таблице векторов прерываний**, расположенной в памяти. При возникновении прерывания или исключения по его номеру в таблице векторов прерываний определяется адрес соответствующей процедуры обработки прерывания или исключения, к которой осуществляется переход. Рассмотрим более подробно, каким образом выполняется вызов и возврат из обработчика прерываний или исключений.

Механизмы обработки прерываний и исключений во многом схожи, но есть некоторые отличия, связанные с возвратом из обработчика. Механизм обработки прерываний включает в себя следующие этапы:

1. Установление факта прерывания.

2. Запоминание в стеке состояния прерванного процесса, которое определяется содержимым регистра флагов *PSW*, счётчика команд *IP*, сегментного регистра *CS*. При необходимости, также запоминается содержимое регистров, которые будут использоваться процедурой прерывания и, следовательно, изменяться. Некоторые типы исключений и прерываний также помещают в стек код ошибки для того, чтобы диагностировать причину, вызвавшую исключение.

3. Определение адреса процедуры обработки прерывания по номеру прерывания в таблице векторов прерываний и осуществление перехода к этому обработчику путём загрузки адреса в регистры *CS* и *IP*.

4. Обработка прерывания. Процедура обработки прерывания выполняет свои команды.

5. Восстановление состояния прерванной программы. После успешного выполнения процедуры обработки прерывания при достижении команды *IRET* (этой командой завершаются обработчики прерываний) из стека восстанавливается старое содержимое сохранённых в нём регистров (старое состояние), в т.ч., и **адрес возврата** – значения регистров *CS* и *IP*.



6. Возврат в прерванную программу. На основании адреса возврата осуществляется переход к прерванному процессу. Возврат должен осуществляться на команду, следующую за командой, выполненной до возникновения прерывания. Процедура обработки прерываний, обладающая таким свойством, называется *прозрачной*.

При возникновении сбоя адрес возврата является адресом команды, вызвавшей сбой, поэтому возврат осуществляется снова к этой команде для попытки её повторного выполнения (рестарта). Обработка ловушек аналогична обработке прерываний. При аварийных завершениях вычислительный процесс завершается без возможности восстановления исходного состояния программы, в которой произошло данное исключение.

Поскольку сигналы прерываний возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов прерываний, которые могут быть обработаны только последовательно. Для этого прерываниям присваиваются приоритеты. Существуют две дисциплины обслуживания приоритетных прерываний: 1) прерывание с более высоким приоритетом может прервать обработку прерывания с более низким приоритетом; 2) прерывание с более низким приоритетом обслуживается до конца, после чего обрабатывается прерывание с более высоким приоритетом.

## **2.2. Организация работы с файлами**

Язык ассемблера не содержит средств для работы с файлами. При возникновении такой необходимости программа должна включать в себя фрагменты кода, в которых производится обращение к средствам операционной системы (ОС), которые осуществляют взаимодействие с файловой системой. Современному программисту часто приходится сталкиваться с необходимостью программирования под ОС MS DOS. Средства работы с файлами этой ОС в плане совместимости поддерживаются различными реализациями Windows. Можно выделить четыре аспекта работы с файлами из программ на ассемблере:

- 1) работа с системой файлового ввода-вывода MS DOS, использующей короткие имена;
- 2) работа с системой файлового ввода-вывода MS DOS, использующей длинные имена файлов (длиной до 255 символов);
- 3) работа с системой файлового ввода-вывода Win32;
- 4) использование файлов особого вида, поддерживаемых Win32.

В дальнейшем будут рассмотрены основные функции работы с файлами под управлением MS DOS, использующей короткие имена (до 8 символов). С одной стороны, использование этих функций не требует специальной подготовки и знаний работы под Win32, с другой – даёт достаточно полное представление об особенностях работы с файлами из программ на ассемблере.

### **2.2.1. Управление дисками и каталогами**

Поверхность диска разделена на ряд концентрических колец (дорожек), которые, в свою очередь, делятся радиально на сектора. Дисковые сектора определяются магнитной информацией, которую записывает утилита форматирования диска. Для всех типов дисков в MS DOS размер сектора равен 512 байт. Файл располагается на таком количестве секторов, которое необходимо для его полного размещения. Диск использует **таблицу размещения файлов (File Allocation Table – FAT)** для отведения дискового пространства файлам и хранения информации о свободных секторах. *FAT* хранит информацию о каждом кластере секторов на диске. **Кластер** – это группа стандартных секторов размером 512 байт. Каждая позиция в *FAT* соответствует определённой позиции кластера на диске. Обычно файл занимает несколько кластеров, и запись в каталоге файлов содержит номер стартового кластера, в котором находится начало файла. Просмотрев позицию *FAT*, соответствующую первому кластеру, MS DOS находит номер кластера, в котором хранится следующая порция файла и т.д. по цепочке.

Каждый диск имеет один корневой каталог, с которого начинается поиск всех каталогов. Корневой каталог может содержать элементы, указывающие на подкаталоги, которые, в свою очередь, могут содержать ссылки на другие подкаталоги. Корневой каталог всегда расположен в определённых секторах диска. Подкаталоги хранятся как обычные дисковые файлы и могут располагаться в любом месте диска. Каталоги имеют различные размеры в зависимости от размера диска и его разбиения на разделы. Как корневой каталог, так и подкаталоги используют 32 байта для хранения информации об одном файле. Описание структуры 32-байтового поля (дескриптора) файла представлено в табл. 2.1.

Точка между именем файла и его расширением не хранится. Все поля выровнены по левой границе, а пустые байты заполняются пробелами. Атрибут файла определяет, является ли он скрытым, защи-

щённым от записи и т.п. Он также определяет специальные элементы каталога: подкаталоги, метка тома. Информация о времени и дате упакована, поэтому для чтения этих значений требуются битовые операции.

Таблица 2.1. *Описание структуры дескриптора файла*

Номера байтов	Назначение байтов
0-7	Имя файла
8-10	Расширение файла
11	Атрибут файла
12-21	Зарезервировано
26-27	Начальный кластер
38-31	Размер файла

Начальный кластер указывает на позицию в таблице *FAT*. Поскольку файл обычно не целиком занимает последний отведённый ему кластер, то в поле «Размер файла» хранится точная длина файла в байтах.

Рассмотрим несколько функций для работы с дисками и каталогами.

### ***Установка/ проверка дискового накопителя по умолчанию.***

Программы могут экономить часть работы, назначая дисковый накопитель, на котором хранятся каталоги и файлы данных, по умолчанию.

1. Для установки дискового накопителя по умолчанию необходимо в регистр *АН* записать функцию *0Eh*, в регистр *DL* поместить номер диска (*00h* – *A*, *01h* – *B* и т.д.), вызвать прерывание *21h*. Эта функция возвращает в регистре *AL* максимально возможный в данной системе номер диска.

2. Функция *19h* прерывания *21h* сообщает о том, какой дисковый накопитель установлен по умолчанию, возвращая номер диска в регистре *AL* (*00h* – *A*, *01h* – *B* и т.д.). Входных регистров для данной функции нет.

### ***Определение доступного дискового пространства.***

Программа должна контролировать доступное дисковое пространство и сообщать пользователю о нехватке места. В этом случае пользователь может выйти из программы и устранить ошибку без потери информации.

Для получения информации о свободном дисковом пространстве необходимо в регистр *АН* записать функцию *36h*, в регистр *DL* по-

местить номер диска ( $00h$  – текущий накопитель,  $01h$  –  $A$  и т.д.), вызвать прерывание  $21h$ . При возврате регистр  $AX$  содержит или код ошибки  $FFFFh$ , если в регистре  $DL$  был указан неправильный номер устройства, или количество секторов в одном кластере, если ошибки не возникло;  $BX$  содержит число свободных кластеров;  $CX$  – размер сектора в байтах,  $DX$  – общее число кластеров на диске. Используя эту информацию, можно посчитать свободное пространство на диске ( $AX \times BX \times CX$ ) и полный объём диска ( $AX \times CX \times DX$ ).

### ***Создание/удаление подкаталога.***

Программа может создавать и удалять подкаталоги только при соблюдении определённых условий. Для создания подкаталога необходимо, чтобы было хотя бы одно пустое место в корневом каталоге. Для удаления подкаталога необходимо, чтобы он не содержал ссылок на другие подкаталоги и файлы (был пустым). Кроме того, нельзя удалить текущий каталог, в котором по умолчанию выполняются все операции над подкаталогами. Также нельзя удалить корневой каталог.

1. Для создания подкаталога необходимо, чтобы пара регистров  $DS:DX$  указывала на строку, содержащую имя накопителя и путь к каталогу, в котором нужно создать подкаталог. Строка должна заканчиваться байтом *ASCII 0*. Последнее имя пути – имя создаваемого подкаталога. Все перечисленные имя каталогов до создаваемого нового должны существовать. Далее в регистр  $AH$  требуется поместить функцию  $39h$  и вызвать прерывание  $21h$ . Если указан правильный путь, то будет создан новый каталог. В противном случае будет установлен флаг переноса  $CF$ , а в регистре  $AX$  будет содержаться код ошибки:  $3$  – несуществующий путь;  $5$  – доступ запрещён.

2. Для удаления подкаталога необходимо, чтобы пара регистров  $DS:DX$  указывала на строку, содержащую путь к удаляемому подкаталогу. Затем в регистр  $AH$  требуется поместить функцию  $3Ah$  и вызвать прерывание  $21h$ . Если указан правильный путь, то будет удалён заданный подкаталог. В противном случае будет установлен флаг переноса  $CF$ , а в регистре  $AX$  будет содержаться код ошибки:  $3$  – несуществующий путь;  $5$  – доступ запрещён;  $10h$  – попытка удаления текущего каталога.

### ***Получение/ изменение текущего каталога.***

**Текущий каталог** – это каталог, в котором MS DOS ищет файл, если к нему не указан путь. Если не установлено противного, то текущим является корневой каталог.

1. Чтобы определить текущий каталог, надо в регистр *АН* поместить функцию *47h*, в регистр *DL* – номер накопителя (*00h* – текущий накопитель, *01h* – *A* и т.д.); пара регистров *DS:SI* должна указывать на 64-байтовый буфер для записи полного пути от корневого каталога. Если был указан несуществующий накопитель, то будет установлен флаг переноса *CF*, а в регистре *AX* будет содержаться код ошибки: *0Fh*. Если ошибок не возникло, то данная функция возвращает строку, которая начинается с имени первого подкаталога пути (имя диска и символ «\» не указываются). Байт *ASCII 0* сигнализирует о конце строки.

2. MS DOS позволяет установить текущий каталог. Для этого пара регистров *DS:DX* должна указывать на путь к подкаталогу (как описано выше при создании и удалении подкаталогов); затем в регистр *АН* поместить функцию *3Bh* и вызвать прерывание *21h*. Если указан правильный путь, то указанный подкаталог будет установлен как текущий. В противном случае будет установлен флаг переноса *CF*, а в регистре *AX* будет содержаться код ошибки: *3* – несуществующий путь.

#### ***2.2.2. Подготовка файлов к операциям чтения/записи***

В программах, написанных на языках высокого уровня, вся подготовительная работа при операциях с файлами выполняется автоматически. При использовании языка ассемблера требуется создание специальных областей данных, которые используются при операциях ввода-вывода. Для доступа к файлам используется метод дескриптора файла. При доступе к файлам MS DOS автоматически создаёт область данных для файла, затем создаёт уникальный 16-битовый код номера файла. Впоследствии этот «номер» используется функциями DOS для идентификации того открытого файла, с которым производится операция. Прежде чем использовать файл в программе, его необходимо открыть. Если файл не существует, то перед открытием его нужно создать. При удалении файла соответствующий элемент каталога на самом деле не удаляется, он становится недействующим за счёт замены первого байта элемента (первого символа имени файла). Впоследствии этот элемент может быть перезаписан при создании нового

файла. Также вносятся изменения в *FAT*, чтобы сектора, занятые удаленным файлом, были доступны для других файлов. Само содержимое при этом не стирается.

### ***Создание/удаление файла.***

1. Можно создать файл, не помещая в него никакой информации. В этом случае создается элемент каталога, а длина файла устанавливается равной 0.

Функция *3Ch* (пересылается в регистр *AH*) прерывания *21h* создает и открывает новый файл. Регистры *DS:DX* должны указывать на строку, представляющую путь к файлу и имя файла в коротком формате и заканчивающуюся *ASCII*-кодом 0. Если файл создается не накопителе, принятом по умолчанию, то в строку включается имя диска. В регистр *CX* помещается байт атрибутов файла, которые представлены в табл. 2.2. Для создания обычного файла с регистр *CX* следует поместить 0.

Таблица 2.2. *Описание структуры регистра атрибутов*

Номера битов	Назначение битов
0	=1 – файл только для чтения
1	=1 – скрытый файл
2	=1 – системный файл
3	=1 – создаваемый элемент является не файлом, а меткой тома; =0 – для создания файла
4	=1 – создаваемый элемент является подкаталогом; =0 – для файла
5	=1 – файл был изменён с даты последней архивации; =0 – файл не был изменён
6-15	=0 – резервные биты

Бит 5 – это архивный бит, используемый программами *BACKUP* и *RESTORE MS DOS*. Этот бит сбрасывается в 0 после архивации и устанавливается в 1, если с файлом снова работали.

При успешном выполнении флаг переноса *CF* будет равен 0, а в регистре *AX* будет возвращён дескриптор файла. В противном случае флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 3 – не найден путь, 4 – нет свободного дескриптора файла, 5 – отказ в доступе.

Если в каталоге уже существует файл с таким именем, он обрезается до нулевой длины и тем самым разрушается. Для более «мягкого» создания и открытия (без ущерба прежнему содержимому) файла

можно использовать функцию *5Bh* прерывания *21h*. Её действие аналогично предыдущей функции; однако, если файл с таким именем существует, она вернёт в регистре *AX* код ошибки *50h*. В этом случае можно перейти к открытию файла.

Также в MS DOS (с версии 3.0) имеется функция *5Ah* прерывания *21h* для создания временного «безымянного» файла. В этом случае сама ОС генерирует имя для файла и проверяет, что такого файла ещё нет в каталоге. Регистры *DS:DX* должны указывать на *ASCII*-строку с путём, заканчивающимся символом «\» и 13 дополнительными нулевыми байтами. Завершается строка *ASCII*-кодом 0. В регистр *CX* помещается байт атрибутов файла, которые представлены в табл. 2.2. После успешного выполнения флаг переноса *CF* будет равен 0, а в регистре *AX* будет возвращён дескриптор файла. Произвольное имя файла добавляется к концу строки пути. При возникновении ошибок флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 3 – не найден путь, 4 – нет свободного дескриптора файла, 5 – отказ в доступе. Файл, созданный этой функцией, не уничтожается автоматически; программа его должна удалить, используя соответствующую функцию.

2. Для удаления файла используется функция *41h* (пересылается в регистр *AH*) прерывания *21h*. Регистры *DS:DX* должны указывать на *ASCII*-строку с путём к удаляемому файлу, заканчивающуюся 0. Если при выполнении функции ошибок не возникло, то указанный файл будет удалён. В противном случае флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 2 – не найден файл, 3 – не найден путь, 5 – отказ в доступе.

Следует отметить, что данная функция не позволяет удалять файлы с атрибутом «только для чтения». В этом случае предварительно необходимо изменить атрибуты удаляемого файла.

### ***Открытие/ закрытие файла.***

1. При открытии файла создаются небольшие блоки памяти, в которые помещается информация о файле. Они будут служить буфером, через который данные будут передаваться между памятью и файлом. Языки высокого уровня создают такой буфер автоматически, язык ассемблера – нет.

При открытии файла проверяется его наличие в каталоге. Если файл найден, ОС MS DOS берёт информацию из каталога о размере и

дате создания файла. По умолчанию в MS DOS может быть одновременно открыто не более 8 файлов.

Для открытия файла используется функция *3Dh* прерывания *21h*. Регистры *DS:DX* должны указывать на *ASCII*-строку, содержащую путь к файлу и его имя, включая имя накопителя, если это необходимо. Вся строка должна быть не длиннее 63 байт и завершаться символом *ASCII 0*. В регистр *AL* помещается код доступа: *0* – открытие файла для чтения, *1* – открытие файла для записи, *2* – открытие файла для чтения/ записи. Если файл открыт успешно, флаг переноса *CF* будет равен *0*, а в регистре *AX* будет возвращён 16-битовый номер файла, по которому файл идентифицируется. Файловый указатель устанавливается на начало файла. Если при открытии файла произошла ошибка, то флаг переноса устанавливается в *1*, а в регистре *AX* возвращается код ошибки: *2* – не найден файл, *4* – открыто слишком много файлов, *6* – ошибка диска, *12* – ошибка кода доступа.

Данная функция позволяет также открывать скрытые файлы.

2. При закрытии файла ОС обновляет информацию в каталоге. Если перед завершением программы не закрыть файл, это может привести к потере данных. Эта функция является необязательной, поскольку функция *4Ch*, которая завершает программу, в числе прочих действий выполняет и закрытие всех файлов.

Для закрытия файла используется функция *3Eh* прерывания *21h*. В регистр *BX* помещается номер (дескриптор) файла, полученный при его открытии. Если файл закрыт успешно, то флаг переноса *CF* будет равен *0*. В противном случае, флаг переноса устанавливается в *1*, а в регистре *AX* возвращается код ошибки: *6* – указан неверный номер (дескриптор) файла.

### ***Получение/ изменение атрибутов файла.***

1. Для получения значений атрибутов файла в регистр *AX* помещается функция *43h*, в регистры *DS:DX* – указатель на строку с путём к файлу и именем файла, завершающуюся символом *ASCII 0*. При успешном выполнении функция устанавливает флаг переноса *CF* в *0*, а в регистре *CX* возвращается слово атрибутов файла (см. табл. 2.2). В противном случае, флаг переноса устанавливается в *1*, а в регистре *AX* возвращается код ошибки: *1* – неверное значение в регистре *AL*, *2* – не найден файл, *3* – указан неверный путь, *5* – доступ запрещён.

2. Для установки новых атрибутов файла используется подфункция *01h* (помещается в регистр *AL*) функции *43h* (помещается в *AX*)



прерывания *21h*. В регистр *CX* помещается новое слово атрибутов файла, в регистры *DS:DX* – указатель на строку с путём к файлу и именем файла, завершающуюся символом *ASCII 0*. В случае успеха флаг переноса *CF* равен нулю, а для указанного файла устанавливаются новые атрибуты. В противном случае, флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 1 – неверное значение в регистре *AL*, 2 – не найден файл, 3 – указан неверный путь, 5 – доступ запрещён.

### ***Переименование файла.***

Для переименования файла используется функция *56h* прерывания *21h*. Регистры *DS:DX* содержат указатель на строку с путём к файлу и именем существующего файла, завершающуюся символом *ASCII 0*; регистры *ES:DI* – указатель на строку с путём к файлу и именем нового файла, завершающуюся символом *ASCII 0*. Имена накопителей, если они присутствуют, должны совпадать. Если пути к файлам не совпадают, то файл переносится в новый подкаталог. Для переноса файла без переименования в другой подкаталог, необходимо указать одинаковые имена существующего и нового файлов, но разные пути. При успешном выполнении функции флаг переноса *CF* равен 0, а файл переименовывается. В противном случае, флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 2 – не найден файл, 3 – указан неверный путь, 5 – доступ запрещён, *11h* – имена накопителей для старого и нового файлов не совпадают.

### ***Чтение/ установка даты и времени последней модификации файла.***

1. Получить дату и время последней модификации файла можно с помощью функции *57h* прерывания *21h*. В регистр *BX* помещается номер (дескриптор) ранее открытого файла. Если ошибок не возникло, то флаг переноса *CF* равен 0, в регистре *CX* возвращается время, в регистре *DX* – дата последней модификации файла. В противном случае, флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 1 – недопустимое значение в *AL*, 6 – недопустимый номер файла.

Время и дата возвращаются в следующих форматах, представленных в табл. 2.3.

2. Для установки времени и даты последней модификации файла используется подфункция *01h* (помещается в регистр *AL*) функции *57h* (помещается в *AH*) прерывания *21h*. В регистр *BX* помещается

номер (дескриптор) ранее открытого файла, в *CX* – новое время, в *DX* – новая дата (в форматах, указанных в табл. 2.3). Если ошибок не возникло, то флаг переноса *CF* равен 0, а для указанного файла устанавливаются новые значения последней модификации. В противном случае, флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: 1 – недопустимое значение в *AL*, 6 – недопустимый номер файла.

Таблица 2.3. *Форматы времени и даты модификации файла*

Время		Дата	
Биты	Описание	Биты	Описание
15-11	Часы (0-23)	15-9	Год
10-5	Минуты	8-5	Месяц
4-0	Секунды	4-0	День

### 2.2.3. Чтение, запись, позиционирование в файле

Имеются два основных метода доступа к файлам: *последовательный* и *прямой*. Однако и последовательные файлы, и файлы прямого доступа, на диске они хранятся одинаково: как непрерывная последовательность байтов. Реально эти два типа файлов различаются по расположению данных в них и по методу доступа к этим данным.

*Последовательные* файлы помещают элементы данных один за другим независимо от их длины, разделяя эти элементы парой символов: возвратом каретки (*ASCII 13*) и переводом строки (*ASCII 10*). Языки высокого уровня вставляют эти символы автоматически; программы на языке ассемблера должны сами заботиться о вставке этих символов после записи каждой переменной в файл. В последовательных файлах могут храниться и числа, и строки. Числа по соглашению записываются в строковом виде. Поскольку элементы данных имеют переменную длину, то невозможно узнать, где в файле расположен определённый элемент. Для поиска нужного элемента программа должна читать файл сначала, отсчитывая нужное число пар «возврат каретки/ перевод строки». По этой причине файлы такого формата называются последовательными. Как правило, с диска в память передаётся весь такой файл.

*Файлы прямого доступа* заранее отводят фиксированное место под каждый элемент данных. Если какой-то элемент данных не занимает всё отведённое пространство, остаток заполняется пробелами. Поскольку каждый элемент занимает одинаковое число байт, то можно легко вычислить местоположение конкретного элемента. Как пра-

вило, связанный набор данных группируется в запись. Каждая запись содержит несколько полей, которые создают набор номеров байтов, начиная с которых пишутся данные элементы. Каждая запись следует непосредственно за предшествующей безо всяких ограничителей типа «возврат каретки/ перевод строки». При этом записи можно делать в любом порядке. Файлы прямого доступа остаются на диске. В памяти присутствуют только отдельные записи, с которыми в данный момент времени идёт работа.

Система хранит файловый указатель для каждого буфера файла. Он указывает на  $n$ -ый байт файла, с которого будет начинаться следующая операция чтения или записи. Когда данные добавляются к последовательному файлу, то указатель первоначально устанавливается на конец файла. При чтении файловый указатель устанавливается на начало файла, последовательно перемещаясь от одного элемента к другому. При обращении к записи в файле прямого доступа положение записи вычисляется в виде смещения относительно начала файла, и указатель устанавливается равным этому значению. Затем нужная запись читается или пишется. Обычно за файловым указателем следит ОС. Однако программа сама может управлять им для своих специальных нужд.

### ***Установка текущей файловой позиции.***

Чтение – запись в файле производится с текущей файловой позиции, на которую указывает файловый указатель.

Установить текущую файловую позицию можно с помощью функции  $42h$  (помещается в регистр  $AH$ ) прерывания  $21h$ . В регистр  $BX$  помещается дескриптор файла, полученный при его открытии. Пара регистров  $CX:DX$  содержит информацию о количестве байт, на которое нужно передвинуть указатель (т.е. смещение новой позиции в файле относительно начальной), которое вычисляется по формуле:  $CX \cdot 65536 + DX$ . В регистр  $AL$  помещается начальное положение в файле, относительно которого производится операция чтения/ записи:  $00h$  – смещение (беззнаковое значение в  $CX:DX$ ) относительно начала файла;  $01h$  – смещение (значение со знаком в  $CX:DX$ ) относительно текущей позиции в файле;  $02h$  – смещение (значение со знаком в  $CX:DX$ ) относительно конца файла.

Если функция выполнена успешно, то флаг переноса  $CF$  равен 0, а пара регистров  $DX:AX$  содержит значение новой позиции в байтах относительно начала файла. В противном случае, флаг переноса уста-

навливается в 1, а в регистре  $AX$  возвращается код ошибки: 1 – недопустимое значение в  $AL$ , 6 – недопустимый номер файла.

Методы позиционирования, заданные значением в регистре  $AL$ , по-разному трактуют величину в паре регистров  $CX:DX$ . Если  $AL=00h$ , то метод позиционирования понимает значение в  $CX:DX$  как абсолютное. Два других метода ( $AL=01h$  и  $AL=02h$ ) рассматривают значение в  $CX:DX$ , как значение со знаком. Необходимо быть внимательным при выполнении операции позиционирования, чтобы избежать последующих ошибок при операциях чтения/ записи:

- значение в  $CX:DX$  указывает на позицию перед началом файла – в этом случае последующая операция чтения/ записи будет выполнена с ошибкой;

- значение в  $CX:DX$  указывает на позицию за концом файла – в этом случае последующая операция чтения/ записи приведёт к расширению файла в соответствии со значением в  $CX:DX$ .

Например, чтобы сдвинуть указатель на конец файла, в регистр  $AL$  надо поместить значение  $02h$ , а содержимое регистров  $CX$  и  $DX$  обнулить. Чтобы сдвинуть файловый указатель на начало файла, надо в регистр  $AL$  поместить величину  $00h$ , а содержимое регистров  $CX$  и  $DX$  обнулить.

### ***Чтение из файла и запись данных в файл последовательного доступа.***

1. Функция  $3FH$  прерывания  $21h$  позволяет читать данные из файла последовательно. Предварительно файл должен быть открыт; при этом файловый указатель автоматически устанавливается на первый байт файла. Файловый указатель уникален для каждого файла: операции над другими файлами не меняют его позицию. Дескриптор открытого файла помещается в регистр  $BX$ , а требуемое для чтения число байтов в регистр  $CX$ . Программа должна отвести место под временный буфер, в который будут читать данные из файла. Такой буфер можно создать прямо в сегменте данных программы. Тогда пара регистров  $DS:DX$  должна указывать на этот буфер.

Если операция чтения выполняется без ошибок, то флаг переноса равен 0, а в регистре  $AX$  возвращается число реально прочитанных байтов. Если  $AX$  равен 0, то достигнут конец файла. Если при чтении произошла ошибка, то в  $AX$  возвращается код ошибки: 5 – ошибка оборудования, 6 – неверный дескриптор файла.

2. Необходима внимательность при открытии последовательного файла для записи данных. Поскольку та же самая функция используется для записи в файл прямого доступа, то при закрытии файла его длина не устанавливается равной последней позиции файлового указателя.

– При открытии последовательного файла для перезаписи надо использовать функцию *3Ch* прерывания *21h*, описанную ранее. Эта функция обычно создаёт новый файл, но, если он существует, то обрезаётся до нулевой длины, т.е. файловый указатель устанавливается равным 0. В регистр *BX* помещается номер файла, в регистр *CX* – число записываемых байтов. Пара регистров *DS:DX* должна указывать на первый байт буфера записываемых данных. После этого в регистр *АН* помещается функция *40h* и вызывается прерывание *21h*. Если при записи ошибок не произошло, то флаг переноса равен 0, а в регистре *АХ* возвращается число реально записанных данных. В противном случае, в *АХ* возвращается код ошибки: 5 – ошибка оборудования, 6 – неверный дескриптор файла.

– Для добавления записей в последовательный файл его необходимо открыть с помощью функции *3Dh* прерывания *21h*, описанную ранее. Файловый указатель должен быть установлен на конец файла, иначе существующие данные будут перезаписаны (функция *42h* прерывания *21h*, приведённая выше). Далее как в предыдущем пункте.

### ***Чтение из файла и запись данных в файл прямого доступа.***

Файл прямого доступа предполагает, что его данные организованы в виде записей фиксированной длины. Таким образом, положение каждой записи может быть вычислено. ОС автоматически выполняет эти вычисления. Однако любая программа сама может выполнять эту работу, манипулируя файловым указателем.

1. Для считывания записи из файла прямого доступа необходимо вычислить позицию в файле, считать запись и поместить её в память. Далее программа должна разделить запись на поля того же размера, которые были использованы при конструировании записи. Также нужно удалить символы пробела, добавленные при заполнении полей. Далее используется та же функция *3Ch* прерывания *21h* как описано выше.

2. Для записи данных в файл прямого доступа программа должна вычислить позицию в файле, в которую требуется установить файло-

вый указатель (пункт 1 данного раздела). Далее используется та же функция 40h прерывания 21h как описано выше.

#### 2.2.4. Организация поиска файлов

Для поиска файлов в каталогах используются функции 4Eh и 4Fh прерывания 21h.

1. Функция 4Eh прерывания 21h ищет файл с заданным именем. Пара регистров DS:DX должна указывать на путь к файлу. Данная строка может содержать до 63 символов и завершаться символом ASCII 0. В регистр CX необходимо поместить слово атрибутов файла, при этом биты 0 и 5 игнорируются.

Если файл найден, то флаг переноса CF равен 0, а область DTA заполняется информацией о нём (Табл. 2.4). **DTA (Disk Transfer Area – область передачи дисковых данных)** – область переноса данных по умолчанию размером 128 байт, которая выделяется каждой программе и начинается со смещения 80h в префиксе программного сегмента. В противном случае, флаг переноса устанавливается в 1, а в регистре AX возвращается код ошибки: 2 – файл не найден, 3 – несуществующий путь, 12h – больше файлов в каталоге нет.

Таблица 2.4. Структура блока данных в DTA

Смещение	Размер в байтах	Описание
00h	1	Буква логического диска, если бит 7=0, то удалённый диск
01h	11	Поисковый шаблон
0Ch	1	Атрибуты поиска
0Dh	2	Порядковый номер файла в каталоге
0Fh	2	№ кластера начала каталога предыдущего уровня
11h	4	Резерв
15h	1	Атрибуты найденного файла
16h	2	Время создания (модификации) файла
18h	2	Дата создания (модификации) файла
1Ah	4	Размер файла
1Eh	13	Имя файла с расширением в виде ASCII-строки, завершающаяся символом ASCII 0

После анализа данной области в программе принимается решение об окончании или продолжении поиска.

2. В имени файла можно использовать символы подстановки: ? – один любой символ, \* – любое количество любых символов. Тогда поиск следующего появления имени файла проводится с помощью

функции *4Fh* прерывания *21h*. Она подготавливается так же, как и предыдущая функция. Указатель на область *DTA* меняться не должен, т.е. там должен содержаться блок, заполненный данными после вызова функции *4Eh*. Если других совпадений не найдено, то флаг переноса устанавливается в 1, а в регистре *AX* возвращается код ошибки: *12h* – больше файлов в каталоге нет.

3. Для выполнения работы, связанной с файлами, MS DOS предоставляет возможность установить собственную область *DTA*.

С помощью функции *2Fh* прерывания *21h* можно получить адрес области памяти, отведённую под *DTA*. Этот адрес будет сформирован в регистрах *ES:BX*. Выделенную область впоследствии можно сделать текущей областью *DTA*.

Для этого можно воспользоваться функцией *1Ah* прерывания *21h*. Пара регистров *DS:DX* должна указывать на область, которая будет областью *DTA* для последующих файловых операций.

При этом все смещения и данные, формируемые функциями поиска файлов, остаются актуальными.

### 2.3. Управление клавиатурой

Для управления клавиатурой можно использовать функции BIOS (Basic Input-Output System) и функции ОС (рассмотрим на примере MS DOS), поскольку язык ассемблера не имеет собственных средств для работы с клавиатурой.

Клавиатура содержит микропроцессор, который воспринимает каждое нажатие на клавишу и выдаёт скан-код в порт микросхемы связи с периферией. **Скан-код** – это однобайтовое число, младшие 7 бит которого представляют идентификационный номер, присвоенный каждой клавише. Старший бит кода говорит о том, была ли клавиша нажата (равен 1) или освобождена (равен 0). Когда скан-код выдаётся в порт микросхемы связи с периферией, вызывается прерывание клавиатуры (*INT 9*). Процессор прекращает свою работу и выполняет процедуру, анализирующую скан-код. При поступлении кода от клавиши сдвига или переключателя изменение статуса записывается в память – слово состояния клавиатуры, расположенного по адресу *0040h:0017h*. Во всех остальных случаях скан-код трансформируется в код символа, при условии, что он подаётся при нажатии клавиши. В противном случае скан-код отбрасывается. После установки клавиш сдвига и переключателей введённый код помещается в буфер клавиатуры, который является областью памяти, способной запомнить вво-

димые символы, пока программа слишком занята, чтобы обработать их.

Выделяют два типа кодов символов: *ASCII*-коды и расширенные коды.

***ASCII-коды*** – это однобайтовые числа, включающие в себя символы пишущей машинки, ряд специальных букв и символов псевдографики, а также 32 управляющих кода (обычно используются для передачи команд периферийным устройствам). В любом случае каждый из названных кодов имеет соответствующий символ, который может быть выведен на экран. ***Расширенные коды*** присвоены клавишам или комбинациям клавиш, которые не имеют представляющего их символа *ASCII*. Расширенные коды имеют длину 2 байта, причём первый байт всегда равен *ASCII 0*. Второй байт – номер расширенного кода (см. табл. 2.5).

Например, код *0:30* представляет *Alt+A*. Начальный ноль позволяет программе определить, принадлежит ли данный код набору *ASCII* или расширенному набору.

Существует несколько комбинаций клавиш для выполнения специальных функций, они не генерируют скан-коды. Например, «*Ctrl+Break*», «*Ctrl+Alt+Del*». Их нажатие приводит к заранее предопределённым результатам.

### ***Очистка буфера клавиатуры.***

Программа должна очистить буфер клавиатуры перед тем, как выдать запрос на ввод. Таким образом, исключаются посторонние нажатия клавиш, которые могут к тому времени накопиться в буфере. Буфер может накапливать до 15 нажатий на клавишу независимо от того, являются ли они однобайтовыми кодами *ASCII* или двухбайтовыми расширенными кодами. Следовательно, буфер должен отвести 2 байта для каждого нажатия на клавишу. Для *ASCII*-кодов первый байт содержит сам *ASCII*-код символа, а второй – скан-код клавиши. Для расширенных кодов первый байт содержит *ASCII 0*, а второй – номер расширенного кода, который обычно совпадает со скан-кодом клавиши, но не всегда.

Сам буфер клавиатуры организован по принципу кольца, имеет размер 16 байтов и занимает в памяти диапазон адресов с *0040h:001Eh* до *0040h:003Dh*. В ячейке *0040h:001Ah* хранится адрес начала (головы) буфера, в ячейке *0040h:001Ch* – адрес конца (хвоста). Если содержимое этих ячеек равно, то буфер пуст. Когда буфер за-



полнен, новые символы игнорируются. Чтобы разрешить ввод 15 символов, требуется 16-ая пустая позиция, 2 байта которой содержат код возврата каретки (*ASCII 13*) и скан-код клавиши «*ENTER*», равный 28.

Таблица 2.5. Сводная таблица расширенных кодов

Значение 2-го байта	Соответствующие клавиши
15	Shift+Tab
16-25	Alt+Q – Alt+P (верхний ряд букв)
30-38	Alt+A – Alt+L (средний ряд букв)
44-50	Alt+Z – Alt+M (нижний ряд букв)
59-68	Функциональные клавиши F1 – F10
71	Home
72	Cursor-Up (стрелка вверх)
73	PageUp
75	Cursor-Left (стрелка влево)
77	Cursor-Right (стрелка вправо)
79	End
80	Cursor-Down (стрелка вниз)
81	PageDown
82	Ins
83	Del
84-93	F1 - F10+Shift
94-103	F1 – F10+Ctrl
104-113	F1 – F10+Alt
114	Ctrl+PrtSc
115	Ctrl+Cursor-Left
116	Ctrl+Cursor-Right
117	Ctrl+End
118	Ctrl+PageDown
119	Ctrl+Home
120-131	Alt+1 – Alt+= (верхний ряд)
132	Ctrl+PageUp

Функция *0Ch* прерывания *21h* выполняет любую из функций ввода с клавиатуры (будут рассмотрены ниже), но предварительно чистит буфер клавиатуры. Надо просто в регистр *AH* поместить функцию *0Ch*, в регистр *AL* – номер функции для ввода символа с клавиатуры и вызвать прерывание *21h*.

## 2) Проверка наличия символа в буфере клавиатуры.

Можно проверить, был ли ввод с клавиатуры, не удаляя символ из буфера клавиатуры, используя возможности BIOS или DOS.

1. *Использование ОС MS DOS.* Функция *0Bh* (помещается в регистр *AH*) прерывания *21h* возвращает в регистре *AL* значение *0FFh*, когда буфер клавиатуры содержит один или более символов, и *0* – в противном случае (буфер пуст).

2. *Использование BIOS.* Функция *1h* прерывания *16h* позволяет не только проверить наличие символа в буфере, но и показывает, какой именно символ содержится в буфере (для 84-клавишной клавиатуры). Флаг нуля *ZF* сбрасывается в *0*, если буфер пуст, и устанавливается в *1*, в противном случае. В последнем варианте копия символа, находящегося в голове буфера, помещается в регистр *AX*, но сам символ из буфера не удаляется. В *AL* возвращается код символа для однобайтовых символов *ASCII*, иначе, *AL* равен *0* для расширенных кодов, и в регистре *AH* помещается номер расширенного кода.

Если были нажаты дополнительные клавиши на расширенных версиях клавиатур, то при проверке символов этой функцией они удаляются из буфера. Поэтому при работе с расширенными клавиатурами следует использовать функции *11h* (для 102-клавишной) и *21h* (для 122-клавишной) прерывания *16h*, соответственно.

***Ввод символа с клавиатуры с ожиданием и эхом, и запись его в буфер клавиатуры.***

При вводе данных и текста эхо вводимых символов обычно выдаётся на экран. При этом символы возврата каретки и забора переводятся в соответствующие перемещения курсора на экране. Выдача эха осуществляется в той позиции, где предварительно был установлен курсор. Текст автоматически переносится на другую строку при достижении конца текущей строки.

1. *Использование ОС MS DOS.* Функция *1h* прерывания *21h* ожидает ввода символа, если буфер клавиатуры пуст, а затем выводит его в текущую позицию курсора. Введённый символ возвращается в регистре *AL*. Если был введён расширенный код, то *AL* будет содержать *0*. Для получения в *AL* второго байта расширенного кода надо повторить вызов прерывания *21h*.

Эта функция игнорирует нажатие клавиши «*ECS*», но обрабатывает «*Ctrl+Break*» («*Ctrl+C*»), выполняя при этом специальную процедуру обработки.

Для ввода нескольких символов данную функцию необходимо использовать в цикле.

2. *Использование BIOS.* Функция *0h* прерывания *16h* позволяет ввести символ с клавиатуры, если буфер пуст (для 84-клавишной клавиатуры). В остальном её действие аналогично действию описанной в предыдущем пункте функции *1h* прерывания *21h*. Для расширенных клавиатур используются функции *10h* (для 102-клавишной) и *20h* (для 122-клавишной) прерывания *16h*, соответственно.

***Ввод символа с клавиатуры без эха и без ожидания и запись его в буфер клавиатуры.***

Некоторые программы, работающие в режиме реального времени, не могут останавливаться и ждать нажатия клавиши.

*Использование ОС MS DOS.* Функция *6h* прерывания *21h* позволяет ввести один символ с клавиатуры, но, в отличие от предыдущей функции, она не ожидает ввода при отсутствии символа в буфере.

В регистр *AH* помещается функция *6h*, в регистр *DL* – значение *0FFh*, и вызывается прерывание *21h*. Флаг нуля *ZF* устанавливается в 1, если буфер клавиатуры пуст. Если символ принят, то он помещается в регистр *AL*. В случае расширенного кода действия аналогичны как для функции *1h*, рассмотренной выше.

Данная функция не обрабатывает «Ctrl+Break» («Ctrl+C»).

***Ввод символа с клавиатуры без эха с ожиданием, и запись его в буфер клавиатуры.***

Обычно вводимые символы отображаются на экране, но в некоторых случаях это нежелательно (например, для расширенных кодов).

*Использование ОС MS DOS.* Функция *7h* прерывания *21h* аналогична функции *1h* прерывания *21h* за исключением того, что не выводит его на экран в виде эха и не воспринимает сочетание клавиш «Ctrl+Break» («Ctrl+C») как прерывание программы.

Функция *8h* прерывания *21h* аналогична функции *1h* прерывания *21h* за исключением того, что не выводит его на экран в виде эха, но воспринимает сочетание клавиш «Ctrl+Break» («Ctrl+C») как прерывание программы.

***Ввод строки символов с клавиатуры.***

Большинство языков программирования предоставляют возможности для ввода строки символов. Они используют возможности ввода символов с эхопечатью, помещая автоматически введенные символы в буфер оперативной памяти. Конечно же, должна быть выделена память, достаточная для приёма символов строки, и должна запи-

сываться длина строки при вводе. Если этого не сделать, то произойдёт отказ системы типа «переполнение буфера».

Функция *0AH* прерывания *21H* позволяет вводить строку длиной до 254 символов, выдавая эхо на терминал. Эта функция продолжает ввод символов до тех пор, пока не нажата клавиша "возврат каретки" ("*ENTER*"). Пара *DS:DX* указывает на строку, куда помещаются вводимые символы. При вводе первый байт этой позиции должен содержать число байтов, отводимой для этой строки. Если первый байт равен 0, то вызов функции игнорируется и программа продолжает выполнение без ожидания ввода строки. После того, как строка введена, второй байт даёт число реально введённых символов. Сама строка начинается с третьего байта.

Надо отвести достаточно памяти для строки нужной длины, плюс 2 байта для дескриптора строки плюс 1 добавочный байт для символа "Возврат каретки". Код возврата каретки: *ASCII 13*, - вводится как последний символ строки, но не учитывается в результате, который функция помещает во второй байт дескриптора.

Таким образом, для получения 50- символьной строки надо отвести минимум 53 байта памяти, и поместить в первый байт памяти строки число *ASCII 51*. После ввода 50 символов второй байт дескриптора будет содержать *ASCII 50*, а 53-й байт отведённой памяти будет содержать *ASCII 13*. Функция обрабатывает нажатие комбинации клавиш «*Ctrl+Break*» («*Ctrl+C*»), останавливая ввод символов и продолжая выполнение программы.

## **2.4. Управление выводом информации на дисплей**

Видеосистема состоит из двух основных частей: видеоадаптера и монитора (дисплея).

Для управления экраном дисплея можно использовать функции BIOS и функции ОС (рассмотрим на примере MS DOS), поскольку язык ассемблера не имеет собственных средств для работы с экраном.

Все видеосистемы используют буферы, в которые отображаются данные для изображения на экране. Экран периодически обновляется сканированием этих данных. Размер и расположение видеобуферов в разных режимах разное. Когда в буфере хранится несколько образов экрана, то каждый отдельный образ называют **дисплейной страницей**.

Любой дисплейный адаптер несколько текстовых и графических режимов, некоторые из них представлены в табл. 2.6.

При выводе текста на каждую позицию экрана отводится 2 байта: первый байт содержит *ASCII*-код символа и посылает символ на экран; второй – байт атрибутов – содержит информацию о том, как должен быть выведен каждый символ.

Все видеоадаптеры, кроме монохромного, предоставляют набор цветных графических режимов, которые различаются как разрешением, так и числом одновременно выводимых цветов.

При этом текст легко комбинируется с графикой.

#### **2.4.1. Вывод символов на экран в текстовом режиме**

##### **Установка/ проверка режима дисплея.**

1. Для установки режима дисплея можно воспользоваться функцией BIOS *0h* (помещается в регистр *AH*) прерывания *10h*. При этом регистр *AL* должен содержать номер режима (см. табл. 2.6): если седьмой бит регистра *AL* равен 0, то экран очищается, в противном случае, содержимое экрана не меняется.

Таблица 2.6. Примеры некоторых режимов дисплея

Режим дисплея	Разрешение X на Y	Количество цветов	Тип	Поддерживаемые графические карты
0	40x25	2	текстовый	CGA, PCjr, EGA, MCGA, VGA
1	40x25	16	текстовый	CGA, PCjr, EGA, MCGA, VGA
2	80x25	2	текстовый	CGA, PCjr, EGA, MCGA, VGA
3	80x25	16	текстовый	CGA, PCjr, EGA, MCGA, VGA
4	320x200	4	графический	CGA, PCjr, EGA, MCGA, VGA
5	320x200	4 (серый)	графический	CGA, PCjr, EGA, MCGA, VGA
6	640x200	2	графический	CGA, PCjr, EGA, MCGA, VGA
7	80x25	2	текстовый	MDA, EGA, VGA
8	160x200	16	графический	PCjr,
9	320x200	16	графический	PCjr,
10	640x200	4	графический	PCjr,
13	320x200	16	графический	EGA, VGA
14	640x200	16	графический	EGA, VGA
15	640x350	2	графический	EGA, VGA
16	640x350	16	графический	EGA, VGA
17	640x480	2	графический	MCGA, VGA
18	640x480	16	графический	VGA
19	320x200	256	графический	MCGA, VGA
20	640x400	16	графический	Tesmar VGA/AD только

2. Для определения текущего режима дисплея используется функция *0Fh* прерывания *10h* (возможности BIOS). Прерывание возвращает номер режима в *AL*, номер текущей страницы экрана – в *BH*, число символов в строке – в *AH*.

### ***Вывод на экран одного символа.***

Все процедуры для вывода символа на экран BIOS и DOS помещают символ в текущую позицию курсора.

1. *Использование BIOS.* Вывод символа с атрибутами осуществляется функцией *9h* (в *AH*) прерывания *10h*. При этом в регистр *AL* помещается код выводимого символа, в *CX* – число повторений символа, в *BH* – номер дисплейной страницы (обычно 0 – текущая страница), в *BL* – атрибуты символа. Значения цветов атрибута даны в табл. 2.7. Для вывода одного символа содержимое регистра *CX* должно быть равно 1.

Таблица 2.7. *Набор кодов цвета*

Код	Цепочка битов	Цвет
0	0000	Чёрный
1	0001	Синий
2	0010	Зелёный
3	0011	Циан
4	0100	Красный
5	0101	Магента
6	0110	Коричневый
7	0111	Белый
8	1000	Серый
9	1001	Ярко-синий
10	1010	Ярко-зелёный
11	1011	Яркий циан
12	1100	Розовый
13	1101	Яркая магента
14	1110	Жёлтый
15	1111	Ярко-белый

В байте атрибутов за цвет символов отвечают первые 4 бита. Биты 0 – 2 отвечают за комбинацию цветов, бит 3 включает высокую интенсивность цвета выводимого символа. Последующие 3 бита позволяют задать цвет фона: биты 4 – 6 также как и биты цвета символа дают 8 возможных комбинаций цветов. Когда включается бит высокой интенсивности, то добавляются ещё 8 цветов. При обычных обстоятельствах бит 7 включает и выключает мигание символа; если

бит 7 также отвечает за высокую интенсивность, то это позволяет получить 16 цветов и для фона.

После вывода символа курсор автоматически не смещается вправо, таким образом, если вывести в цикле несколько символов с помощью этой функции, то все они будут выведены в одну и ту же позицию курсора. Новый символ будет затирать предыдущий. Кроме того, данная функция не интерпретирует управляющие символы.

2. *Использование BIOS.* Действие функции *0Ah* (в *AH*) прерывания *10h* аналогично рассмотренной выше за исключением того, символ выводится на экран с текущим значением атрибута.

3. *Использование ОС MS DOS.* Функции *2h* и *6h* прерывания *21h* позволяют вывести один символ на экран без атрибутов (белые символы на чёрном фоне). Однако при выводе нескольких символов в цикле они автоматически сдвигают курсор на одну позицию вправо.

И для одной и для другой функции справедливо: в *AH* помещается номер функции, в *DL* – код символа (кроме значения *0FFh* для функции *6h*). При этом функция *2h* распознаёт «*Ctrl+Break*» («*Ctrl+C*»), а *6h* – нет.

Эти функции интерпретируют некоторые управляющие символы.

Следует отметить, что, если вывод на экран символов по их *ASCII*- коду трудностей не составляет, то для вывода чисел необходимо их преобразование. Проще всего вывести на экран числа в диапазоне от 0 до 9: надо просто к *ASCII*-коду символа прибавить значение *30h* и воспользоваться одной из указанных выше функций вывода. Для чисел большей размерности можно воспользоваться алгоритмом выделения отдельных цифр, к которым применить приведённые выше преобразования.

### ***Вывод на экран строки символов.***

*Использование ОС MS DOS.* Функция *9h* прерывания *21h* выводит строку. Пара *DS:DX* должна указывать на первый символ строки. Строка должна заканчиваться символом '\$' (символ '\$' не входит в выводимую строку). Строка может быть любой длины. Функция не переводит автоматически курсор на начало следующей строки после завершения вывода. Чтобы это выполнялось, надо добавить в конец строки символы до символа '\$': *CR* (*0DH*, «возврат каретки») и *LF* (*0AH*, «перевод строки»). Данная функция обрабатывает сочетание клавиш «*Ctrl+Break*» («*Ctrl+C*»). Также эта функция интерпретирует ряд управляющих символов.

### ***Очистка экрана.***

Существует несколько способов очистки экрана.

1. Очистить экран целиком можно путём изменения видеорежима экрана с помощью функции *0h* прерывания *10h*. Данный метод удобно использовать в начале программы, когда требуется установка режима работы видеоадаптера.

2. Можно использовать функции сдвига экрана *6h* и *7h* прерывания *10h* для полной или частичной очистки экрана.

Функция *6h* (пересылается в *AH*) позволяет определить на экране окно, в котором возможно прокрутить определённое число строк вверх. При этом верхние строки исчезают, а снизу добавляются пустые строки.

Функция *7h* (пересылается в *AH*) позволяет определить на экране окно, в котором возможно прокрутить определённое число строк вниз. При этом нижние строки исчезают, а сверху добавляются пустые строки.

Число строк, на которое надо сдвинуть экран вверх или вниз, помещается в регистр *AL*. При *AL = 0* экран очищается. Координаты левого верхнего угла окна помещаются в регистр *CX* (*CH* – строка, *CL* – столбец), координаты правого нижнего угла – в регистр *DX* (*DH* – строка, *DL* – столбец).

### ***2.4.2. Управление курсором***

Курсор служит двум целям: 1) служит указателем места на экране для вывода информации и 2) обеспечивает видимую точку отсчёта на экране для пользователя программы. Если видеоадаптер поддерживает работу с несколькими дисплейными страницами, то каждая страница имеет свой собственный курсор. Информация о курсоре хранится в двухбайтовой переменной: младший байт содержит номер столбца (начиная с 0), старший байт – номер строки (начиная с 0). Абсолютные координаты курсора могут меняться в пределах 25 строк и 80 (иногда 40) столбцов. При этом координаты 0,0 определяют левый верхний угол экрана.

### ***Установка/ чтение позиции курсора.***

1. Функция *2h* (пересылается в *AH*) прерывания *10h* устанавливает курсор, относящийся к указанной дисплейной странице, в абсолютную позицию, которая будет являться начальной для последующего вывода информации. Страницы нумеруются, начиная с 0 (0 –



текущая дисплейная страница). При этом, в регистр *DH* помещается позиция курсора по строке, в регистр *DL* – позиция курсора по столбцу, а в *BH* – номер дисплейной страницы. После обработки прерывания курсор меняет своё положение на экране, если установка курсора относится к текущей активной дисплейной странице.

Если необходимо сдвинуть курсор относительно его текущей позиции, то для этих целей можно использовать ту же функцию.

2. Для получения текущей позиции курсора необходимо в регистр *AH* поместить функцию *3h*, в *BH* – номер дисплейной страницы и вызвать прерывание *10h*. После обработки прерывания в регистре *DH* будет находиться положение курсора по строке, в регистре *DL* – положение курсора по столбцу.

### ***Включение/отключение курсора.***

Ассемблерные программы оставляют курсор включённым до тех пор, пока не указано обратное. ОС не предоставляет специальных средств отключения курсора, но это легко сделать с помощью функции *2h* прерывания *10h*, поместив за пределы экрана, например, в первую позицию 26 – ой строки (координаты *DL* = 0, *DH* = 25). Соответственно, чтобы сделать его снова видимым, надо вернуть курсор в пределы экрана текущей активной страницы.

### ***2.4.3 Вывод точечной графики***

Различные видеоадаптеры:

1. монохромный *MDA – Monochrome Display Adapter*;
2. цветной графический адаптер *CGA – Color Graphics Adapter*;
3. усовершенствованный графический адаптер *EGA - Enhanced Graphics Adapter*;
4. видеографическая матрица *VGA - Video Graphics Array*.

Они по-разному работают с цветом и графикой. Адаптеры EGA, VGA поддерживает работу в монохромном и CGA-совместимом режимах. Более того, эти режимы поддерживаются как на монохромных, так и на цветных мониторах. Обращения к функциям BIOS (прерывание *10h*) совместимы между различными типами IBM – (и совместимых с IBM) адаптеров. Оригинальные программы, позволяющие использовать особенности каждого типа видеоадаптеров, связаны с применением других прерываний и регистров портов адаптеров, и являются достаточно сложными. Поэтому операции, связанные с выводом точечной графики рассмотрим на примере функций прерывания *10h*.

Изображение на экране растрового дисплея формируется посредством группы горизонтальных строк, называемых *растром*. Каждая точка (пиксель) цветного экрана состоит из трёх цветных точек: красной, зелёной, синей (*RGB – Red Green Blue*).

В графическом режиме могут выводиться и символы. Однако они создаются не обычным способом: BIOS вырисовывает их по точкам, не изменяя цвета фона. Поэтому негативное изображение и мигание символов недоступны в графическом режиме. Не выводится и курсор.

### ***Установка цвета фона.***

Функция *0Bh* (пересылается в *AH*) прерывания *10h* позволяет установить цвет фона. Для этого в регистр *BH* записывается 0, в регистр *BL* – номер цвета от 0 до 15 (см. табл. 2.7).

### ***Вывод точки на экран.***

Функция *0Ch* (пересылается в *AH*) прерывания *10h* устанавливает на экране точку. В регистр *CX* необходимо поместить координату по строке (по горизонтали), в *DX* – координату по столбцу (по вертикали). Они отсчитываются от 0. Код цвета помещается в регистр *AL*. При отсутствии ошибок на экране формируется точка указанного цвета в указанной позиции.

### ***Чтение точки с экрана.***

Функция *0Dh* (пересылается в *AH*) прерывания *10h* позволяет прочесть точку для определения её цвета. В регистр *CX* необходимо поместить координату по строке (по горизонтали), в *DX* – координату по столбцу (по вертикали). После обработки прерывания цвет указанной точки возвращается в регистре *AL*.

## **2.5. Управление таймером**

Все IBM PC используют микросхему таймера 8253 (или 8254) для согласования импульсов от микросхемы системных часов. Число циклов системных часов преобразуется в один импульс, а последовательность импульсов подсчитывается для определения времени или импульсы могут быть посланы на громкоговоритель компьютера для генерации звука определённой частоты. Данная микросхема имеет 3 независимых канала (0, 1, 2), каждый из которых может программироваться. Доступ к каналам осуществляется через порты *40h*, *41h*, *42h*, соответственно.

Канал 0 используется системными часами времени суток. Он устанавливается BIOS при старте таким образом, что выдаёт импульсы приблизительно 18,2 раза в секунду. Каждый импульс инициирует прерывание таймера, что увеличивает показание счётчика. Число 0 соответствует полночи 12:00; когда счётчик достигает значения, эквивалентного 24 часам, он сбрасывается в 0. Другое значение времени в течение суток определяется делением показателя счётчика на 18,2 для каждой секунды.

Канал 1 управляет обновлением памяти, поэтому его лучше не трогать.

Канал 2 связан с громкоговорителем компьютера, и он производит простые прямоугольные импульсы для генерации звука.

### ***2.5.1. Работа со счётчиком времени суток***

#### ***Чтение/установка времени.***

MS DOS предоставляет прерывания для чтения и установки времени, производя необходимые преобразования между значением счётчика времени суток и часами – минутами – секундами. Время выдаётся с точностью до одной сотой секунды, но, поскольку счётчик обновляется с частотой в 5 раз меньшей, то показания сотых долей очень приближённые.

1. Функция *2Ch* (пересылается в *AH*) прерывания *21h* позволяет прочесть текущее время системных часов. При этом в регистре *CH* возвращаются часы (в диапазоне 0 – 23), в *CL* – минуты (0 – 59), в *DH* – секунды (0 – 59), в *DL* – сотые доли секунды (0 – 59), в *AL* – номер дня недели (0 – 6, 0 – воскресенье). День недели будет возвращён верно, если была установлена дата.

2. Функция *2Dh* (пересылается в *AH*) прерывания *21h* позволяет установить новое время системных часов. При этом в регистр *CH* помещаются часы (в диапазоне 0 – 23), в *CL* – минуты (0 – 59), в *DH* – секунды (0 – 59), в *DL* – сотые доли секунды (0 – 59).

Если при установке времени ошибок не возникло, то в регистре *AL* возвращается 0, в противном случае – значение *FF*.

#### ***Чтение/установка даты.***

Значение даты хранится в переменной файла COMMAND.COM. Она хранится в формате трёх последовательных байтов, которые содержат день месяца, номер месяца и год месяца (0 соответствует 1980 году).

1. Функция  $2Ah$  (пересылается в  $AH$ ) прерывания  $21h$  позволяет прочитать текущую дату. При возврате в регистре  $CX$  содержится год в диапазоне от 0 до 59, что соответствует годам в интервале от 1980 до 2099. В регистре  $DH$  – номер месяца (1 – 12), в  $DL$  – день (1 – 31).

2. Функция  $2Dh$  (пересылается в  $AH$ ) прерывания  $21h$  позволяет установить новую дату. При этом в регистр  $CX$  помещается год (в диапазоне 0 – 119), в  $DH$  – номер месяца (1 – 12), в  $DL$  – день (1 – 31).

Если при установке даты ошибок не возникло, то в регистре  $AL$  возвращается 0, в противном случае – значение  $FF$ .

### ***Задержка программных операций.***

При реализации задержки в программе посредством пустого цикла может потребоваться много времени для того, чтобы добиться нужного времени задержки. Длительность цикла может меняться в зависимости от применяемого компилятора. Поэтому целесообразно определять время непосредственно по часам. Чтобы обеспечить задержку данной продолжительности, программа должна подсчитать требуемое количество импульсов счётчика времени суток. Например, если необходимо организовать задержку на 5 секунд, то число импульсов счётчика составит  $5 \cdot 18.2 \approx 91$  импульсу. Это значение добавляется к считанному текущему значению счётчика. Затем программа постоянно считывает значение счётчика и сравнивает его с запомненным. При достижении равенства задержка считается выполненной.

Значение счётчика времени суток хранится в четырёх байтах. Два младших байта позволяют осуществлять задержки до одного часа.

Функция  $0h$  (пересылается в  $AH$ ) прерывания  $1Ah$  позволяет прочитать текущее значение счётчика времени суток (в импульсах). При возврате в регистре  $DX$  содержатся два младших байта, в регистре  $CX$  – два старших байта. Для задержек в пределах одного часа старшие байты можно игнорировать и работать только с регистром  $DX$ .

### ***2.5.2. Генерация звука***

Канал 2 микросхемы таймера прямо связан с динамиком компьютера. Когда этот канал программируется в режиме 3, он посылает прямоугольные волны данной частоты. Из-за простоты динамика он сглаживает края прямоугольной волны, получая синусоидальную волну, более приятную для слуха. Микросхема таймера не позволяет менять амплитуду волны, поэтому нельзя менять громкость звука, издаваемого динамиком. Кроме того, сигнал также посылает микросхе-

ма связи с периферией. Комбинируя воздействия этих двух источников, можно получать различные звуковые эффекты.

Если процессор непосредственно управляет динамиком, то для генерации звука аппаратные прерывания должны быть отключены оператором *CLI* перед началом работы со звуком и включены оператором *STI* после окончания работы со звуком. Изменяя частоту (в диапазоне от 37 Гц до 32767 Гц) и длительность (в диапазоне от 0 до 65535 импульсов счётчика времени суток, или используя пустой цикл), можно получать различные звуки. Частоты нот первой октавы приведены в табл. 2.8.

Таблица 2.8. Частоты нот первой октавы

Нота	Частота
До	523,3
Ре	587,3
Ми	659,3
Фа	698,5
Соль	784,0
Ля	880,0
Си	987,7

Частоты на октаву выше можно получить, удваивая эти значения, частоты на октаву ниже приблизительно равны половине указанных в табл. 2.8 значений.

Генерация звука состоит во включении и выключении с желаемой частотой первого бита порта *61h* микросхемы интерфейса с периферией, который связан с динамиком.

Для генерации звука определённой частоты необходимо в порт *42h* (канал 2 микросхемы таймера) переслать желаемую частоту. Поскольку в порт можно за раз записать не более 1 байта, то пересылка частоты выполняется в два этапа (сначала посылается младший байт частоты, а потом старший). Затем в режиме запрещения аппаратных прерываний (команда *CLI*) необходимо отключить динамик от таймера, сбросив в 0 нулевой бит порта *61h*. Потом подключить динамик к таймеру и включить звук, установив в единицу нулевой и первый биты порта *61h*. По истечении требуемой длительности следует отключить звук, следует отключить звук, сбросив в 0 нулевой и первый биты порта *61h*. После работы со звуком необходимо включить режим разрешения аппаратных прерываний командой *STI*.

Следует отметить, что при отключении аппаратных прерываний счётчик времени суток BIOS работать не будет. Поэтому для задания длительности целесообразно использовать пустой цикл.

## **2.6. Управление прерываниями. Написание собственного прерывания**

В некоторых случаях бывает полезным написание собственного прерывания. Например, многие программы конкретного пользователя могут использовать процедуру, выводящую строки на экран вертикально. Вместо того чтобы включать её в каждую программу в качестве процедуры, можно установить её как прерывание, написав соответствующую процедуру.

Краткое описание прерываний таблицы векторов прерываний представлено в табл. А.1 приложения А настоящего практикума (IRQ0 – IRQ15 – это аппаратные прерывания). Как следует из табл. А.1, векторы прерываний в диапазоне 60Н – 67Н отведены для пользовательских прерываний.

Функция *25h* (пересылается в *AH*) прерывания *21h* устанавливает вектор прерывания на указанный адрес. Адреса имеют размер в два слова. Старшее слово содержит адрес сегмента (*CS*), младшее - смещение (*IP*). Чтобы установить вектор, указывающий на одну из процедур пользователя, нужно поместить начальный адрес процедуры в регистр *DS* (через регистр *AX*), а смещение – в регистр *DX*. Номер прерывания, за которым будет закреплена указанная процедура (например, *60h*) помещается в регистр *AL*. Любая процедура прерывания должна завершаться инструкцией *IRET*, которая выталкивает из стека три слова: адрес возврата и регистр флагов. Следует отметить, что функция *25h* автоматически запрещает аппаратные прерывания, поэтому не существует опасности, что в процессе перехода к процедуре произойдёт аппаратное прерывание, использующее данный вектор.

Когда программа завершается, должны быть восстановлены оригинальные векторы прерываний. В противном случае последующая программа может вызвать данное прерывание и передать управление на то место памяти, в котором пользовательской процедуры уже нет. Функция *35h* (пересылается в *AH*) прерывания *21h* возвращает текущее значение вектора прерывания, помещая адрес сегмента в регистр *ES*, а смещение – в регистр *BX*. Перед установкой собственного прерывания рекомендуется получить текущее значение вектора, сохра-

нить эти значения и затем восстановить их с помощью функции *25h* (как сказано выше) перед завершением своей программы

Существуют «ловушки», которых следует избегать при написании собственных прерываний. Например, если программа завершается по нажатию «*Ctrl+Break*», то вектор прерывания не будет восстановлен, если пользователем не предусмотрено противное.

## **2.7. Управление мышью**

Манипулятор «мышь» относится к интерактивным устройствам, обеспечивающим связь человека с ВМ.

Единицей перемещения мыши является «*микки*» - сигналы, которые подсчитывает мышь, а потом через определённые интервалы передаёт драйверу. Последний использует эти сигналы для определения положения мыши, преобразуя их в пиксели на экране. Драйвер автоматически перемещает курсор на экране в соответствии с движением мыши и отслеживает координаты курсора, а также предоставляет пользовательской программе ряд функций по использованию мыши. Эти функции обрабатываются прерыванием мыши *INT 33h*.

### ***Начальная установка драйвера мыши.***

Функция *0h* (пересылается в *АН*) прерывания *33h* производит начальную установку и возвращает информацию о текущем состоянии аппаратных и программных средств мыши. Функция определяет текущий режим экрана, прячет курсор и помещает его в центр экрана, а также задаёт начальные значения внутренним переменным драйвера. В качестве результата в регистре *АХ* возвращается значение *FFFFh* (мышь и драйвер мыши установлены), в регистре *ВХ* – количество кнопок мыши: *0002h* – две кнопки, *0003h* – три кнопки, *0000h* – другое количество кнопок. В противном случае, в регистре *АХ* возвращается значение *0000h* – мышь или драйвер мыши не установлены.

### ***Показ/сокрытие курсора мыши.***

Драйвер мыши поддерживает внутренний счетчик, управляющий видимостью курсора мыши. Функция *2h* уменьшает значение счетчика на единицу, а функция *1h* увеличивает его, но только до значения 0. Если значение счетчика – отрицательное число, он скрыт, если ноль – показан.

1. Чтобы показать курсор необходимо в регистр *АН* переслать функцию *1h* и вызвать прерывание *33h*.

2. Чтобы спрятать курсор необходимо в регистр *АН* переслать функцию *2h* и вызвать прерывание *33h*.

Нужно иметь в виду, что на каждый вызов функции *2h* впоследствии должен быть произведен вызов функции *1h*, для того чтобы восстановить внутреннее значение флажка курсора. Кроме того, при каждом переключении режима экрана функция *2h* вызывается автоматически.

### ***Определение состояния мыши.***

Для определения положения курсора мыши и состояния её кнопок используется функция *3h* (пересылается в *АН*) прерывания *33h*. В качестве результата эта функция возвращает в регистре *ВХ* состояние кнопок мыши, в регистре *СХ* – координату по оси *X*, в регистре *ДХ* – координату по оси *Y*. В регистре *ВХ* первые 3 бита (0 – 2) отвечают за состояние кнопок (см. табл. 2.9), остальные биты не используются.

Таблица 2.9. *Биты состояния кнопок мыши*

Номер бита	Описание
0	Левая кнопка мыши
1	Правая кнопка мыши
2	Средняя кнопка мыши

Если бит установлен (равен 1), то соответствующая кнопка нажата, если сброшен (равен 0) – кнопка отпущена.

Возвращаемые координаты совпадают с пиксельными координатами соответствующей точки на экране в большинстве графических режимов, кроме *4*, *5*, *0Dh*, *13h*, в которых *X* – координату мыши нужно разделить на 2, чтобы получить номер столбца соответствующей точки на экране. В текстовых режимах обе координаты надо разделить на 8, чтобы получить номер строки и столбца соответственно.

### ***Установка/отмена обработчика событий мыши.***

В большинстве случаев указанная выше функция не используется в программах, так как для того, чтобы реагировать на нажатие кнопки или перемещение мыши в заданную область, требуется вызывать это прерывание постоянно, что приводит к трате процессорного времени. Гораздо эффективнее указать драйверу самому следить за её передвижениями и передавать управление в программу, как только выполнится заранее определенное условие (например, нажатие левой кнопки мыши). В этом случае выполнение прикладной программы



временно приостанавливается, и управление передаётся по установленному адресу.

Такую возможность обеспечивает функция *0Ch* (пересылается в *АН*) – установить обработчик событий. Битовая маска определяет условия, вызывающие прерывания. Установленный в 1 бит разрешает прерывание, сброшенный в 0 – запрещает прерывание. При вызове данной функции битовая маска помещается в регистр *CX*; описание битов маски представлено в табл. 2.10.

Таблица 2.10. Описание битов маски

Номер бита	Описание
0	Изменение позиции курсора мыши
1	Нажатие левой кнопки мыши
2	Отпускание левой кнопки мыши
3	Нажатие правой кнопки мыши
4	Отпускание правой кнопки мыши
5	Нажата средняя кнопка мыши
6	Отпускание средней кнопки мыши
7-15	Не используются

Драйвер фирмы Microsoft следит за позицией мыши, а драйверы фирм Logitech и Mouse Systems – за позицией курсора.

Также при вызове функции пара регистров *ES:DX* должна содержать адрес дальнего перехода на процедуру обработки прерываний.

Обработчик событий должен быть оформлен, как дальняя процедура. На входе в процедуру обработчика необходимо передать следующие параметры:

- 1) регистр *AX* содержит условие вызова (биты установлены также как и в маске вызова);
- 2) регистр *BX* содержит состояние кнопок;
- 3) регистры *CX*, *DX* – *X*- и *Y*-координаты курсора мыши, соответственно;
- 4) регистры *SI*, *DI* – счетчики последнего перемещения по горизонтали и вертикали (единицы измерения для этих счетчиков – микро, 1/200 дюйма).

После завершения работы подпрограммы работа прикладной программы возобновляется в той точке, где была прервана.

Перед завершением программы установленный обработчик событий должен быть обязательно удален (вызов функции *0Ch* с *CX=0*), так как иначе при первом же выполнении условия управления будет передано по адресу в памяти, с которого начинался обработчик.

### 3. ПРОЦЕСС АССЕМБЛИРОВАНИЯ И ВЫПОЛНЕНИЯ ПРОГРАММЫ

Для выполнения лабораторных работ и практических заданий необходимо следующее оборудование и программное обеспечение: IBM совместимый персональный компьютер с установленной ОС Windows XP, Windows 7 и выше, виртуальная машина DOSBox, любой текстовый редактор, например, Блокнот, а также программный пакет TASM.

#### 3.1. Получение исполняемого файла

DOSBox – эмулятор для, создающий DOS-окружение, необходимое для запуска старых программ и игр под управлением ОС MS-DOS. Это позволяет играть в такие игры в ОС, не поддерживающих или поддерживающих DOS-программы не полностью, и на современных компьютерах, на которых иначе старые программы могут не работать или работать с ошибками. Эмулятор имеет открытый исходный код и доступен для таких систем, как Linux, FreeBSD, Windows, Mac OS X, iOS, OS/2, BeOS, KolibriOS, Symbian OS, QNX, Android. Также Windows-версия при помощи HX DOS Extender запускается под чистым DOS и, таким образом, DOS эмулируется под DOS. Эмулятор DOSBox является бесплатной программой, и её дистрибутив можно найти в открытых источниках интернета.

Дистрибутив DOSBox устанавливается на компьютер, после чего эмулятор становится доступным для работы. Например, в ОС Windows XP, соответствующий файл появится в меню *Пуск/ Программы*.

В программный пакет TASM входят компилятор, загрузчик, отладчик, а так же необходимые для работы библиотеки. Также может входить файл помощи для работы с отладчиком, однако он не является обязательным. Перечень файлов пакета TASM:

- 1) DPMILOAD.EXE;
- 2) DPMIMEM.DLL;
- 3) DPMI16BI.OVL;
- 4) TASM.EXE;
- 5) TLINK.EXE;
- 6) TD.EXE;
- 7) TDHELP.TDH – необязательно.

Перечисленные выше файлы также можно найти в открытых источниках интернета или получить у преподавателя. Данные файлы

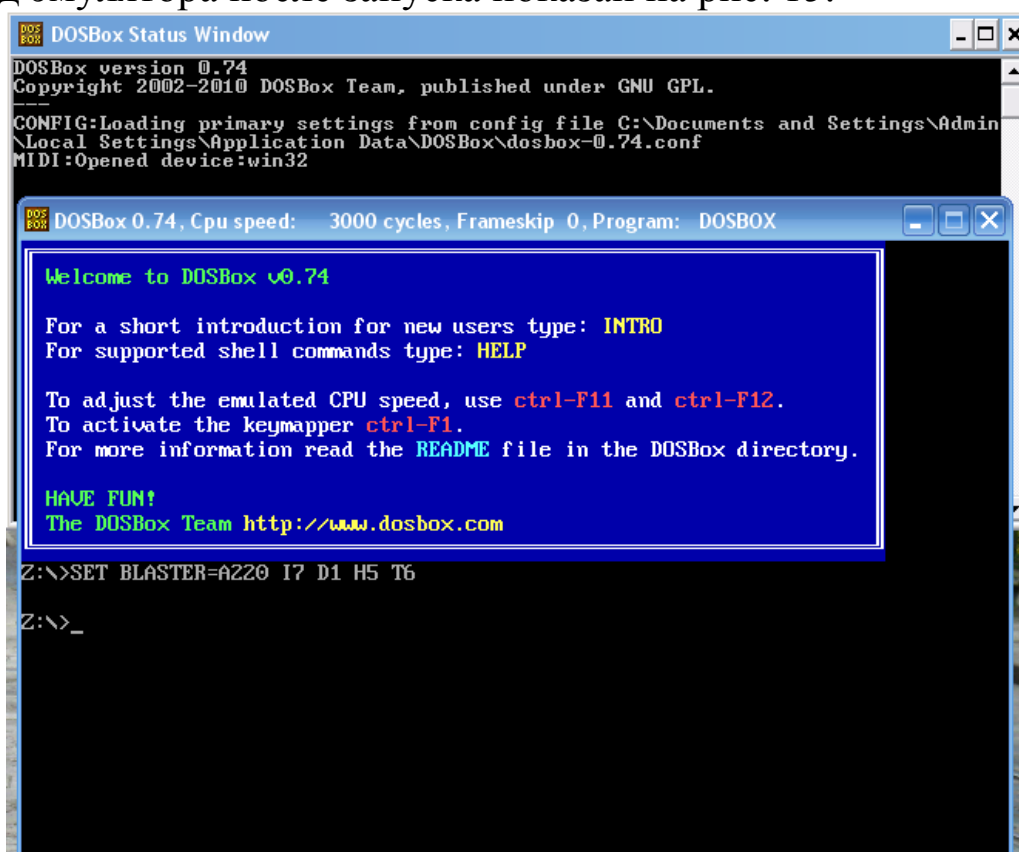
необходимо разместить в отдельной папке и скопировать в рабочую директорию на диске своего компьютера.

Указанный пакет TASM позволяет использовать команды ассемблера для процессора Intel 8086.

Для получения исполняемого файла (с расширением .EXE), необходимо выполнить следующие действия:

1. Создать в любом текстовом редакторе, допустим, в Блокноте, исходную программу на языке ассемблера, и сохранить его как файл с расширением .ASM в рабочую директорию своего компьютера.

2. Запустить эмулятор DOSBox, выбрав соответствующую команду в меню Пуск или ярлык на рабочем столе компьютера. Внешний вид эмулятора после запуска показан на рис. 15.

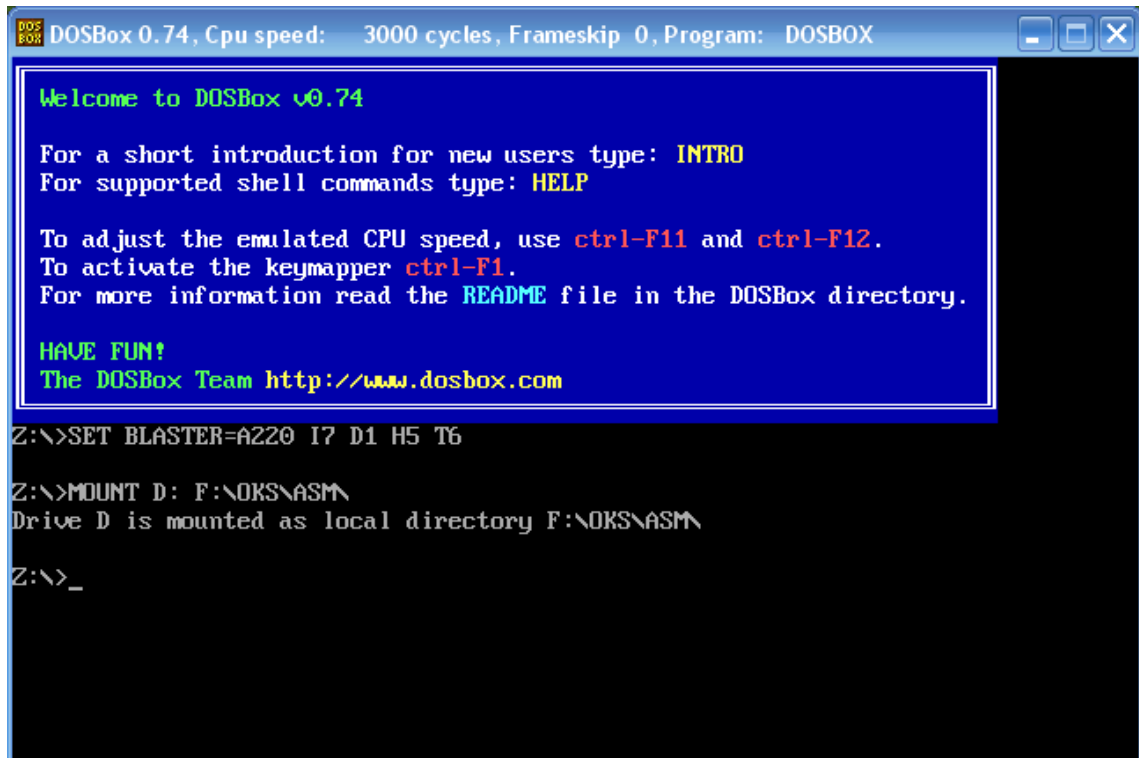


**Рис. 15.** Внешний вид эмулятора DOSBox после запуска

3. Связать диск D: в ОС MS DOS с папкой Windows (или другой ОС), в которой расположены файлы пакета TASM и исходная программа. Желательно, чтобы перечисленный выше семь файлов и исходный файл .ASM находились в одной папке. Например, если пакет TASM находится на F:\OKS\ASM, то в строке приглашения необходимо указать следующее:

**MOUNT D: F:\OKS\ASM\**

Результат выполнения показан на рис. 16.



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

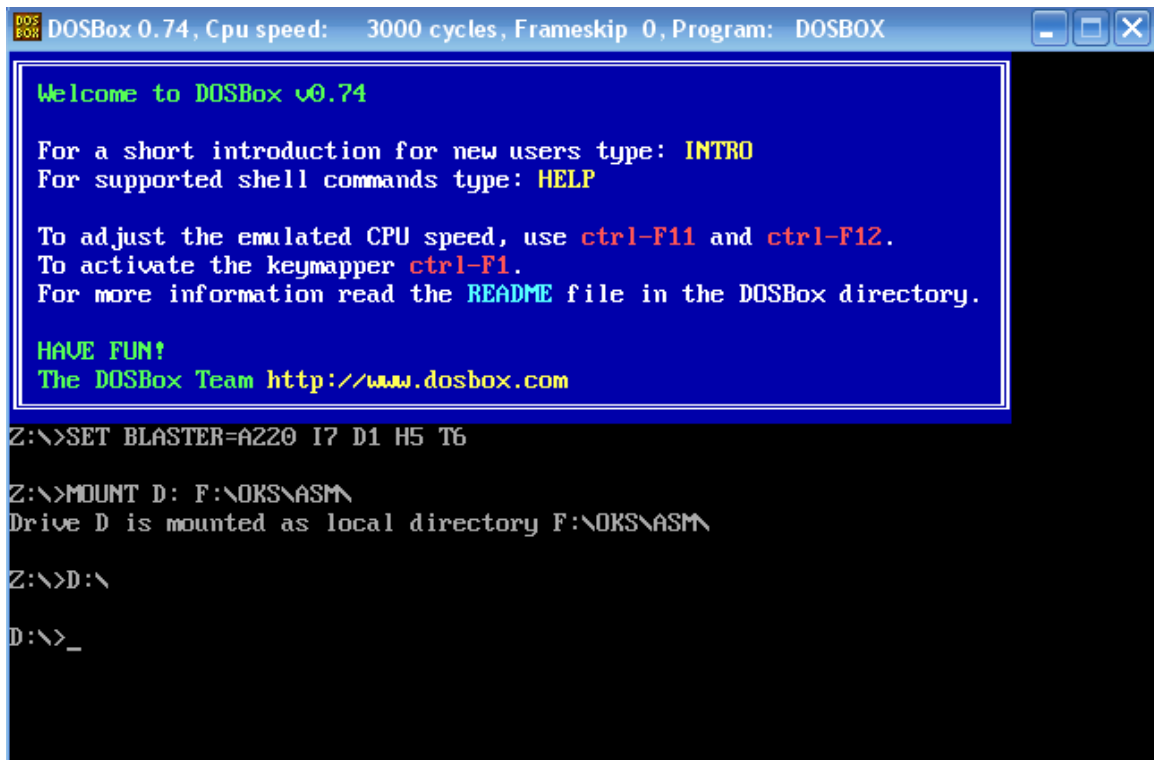
Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT D: F:\OKS\ASM\
Drive D is mounted as local directory F:\OKS\ASM\

Z:\>_
```

*Рис. 16.* Сопоставление директорий

4. Перейти на диск *D:* в MS DOS, что равносильно переходу в соответствующую папку Windows, указанную в команде *MOUNT* (см. рис. 17).



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT D: F:\OKS\ASM\
Drive D is mounted as local directory F:\OKS\ASM\

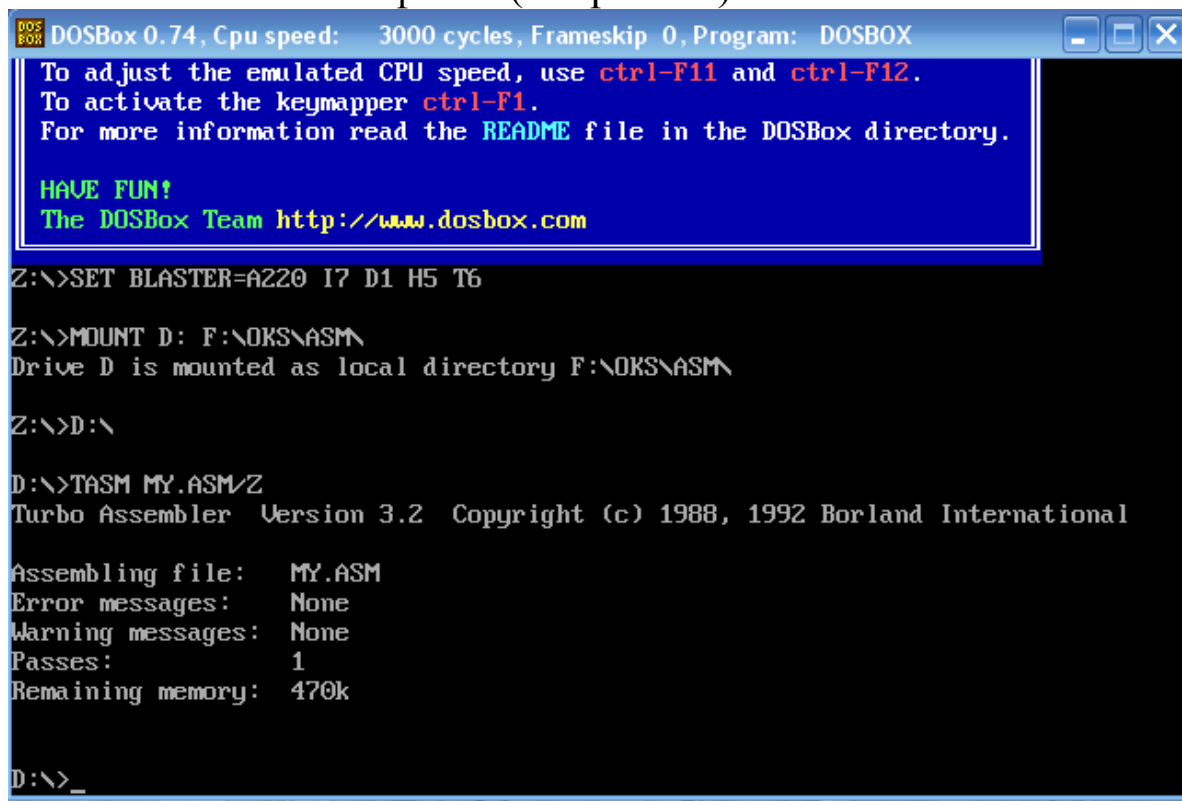
Z:\>D:\
D:\>_
```

*Рис. 17.* Переход в рабочую директорию

5. Странслировать исходный файл с расширением .ASM путём ввода в командной строке следующей команды:

**TASM \Путь\Имя файла.ASM /Z**

Например, если надо странслировать файл MY.ASM, то в команде указывается имя этого файла (см. рис. 18).



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX
To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT D: F:\OKS\ASM\
Drive D is mounted as local directory F:\OKS\ASM\

Z:\>D:\>
D:\>TASM MY.ASM/Z
Turbo Assembler Version 3.2 Copyright (c) 1988, 1992 Borland International

Assembling file: MY.ASM
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 470k

D:\>_
```

**Рис. 18.** Пример трансляции файла MY.ASM

После трансляции на экране появится сообщение (см. рис. 18):

Assembling file: транслируемый файл.

Error messages: сообщения об ошибках. ( None – нет ошибок).

Warning messages: предупреждающее сообщение.

Passes: количество страниц.

Remaining memory: занимаемая память.

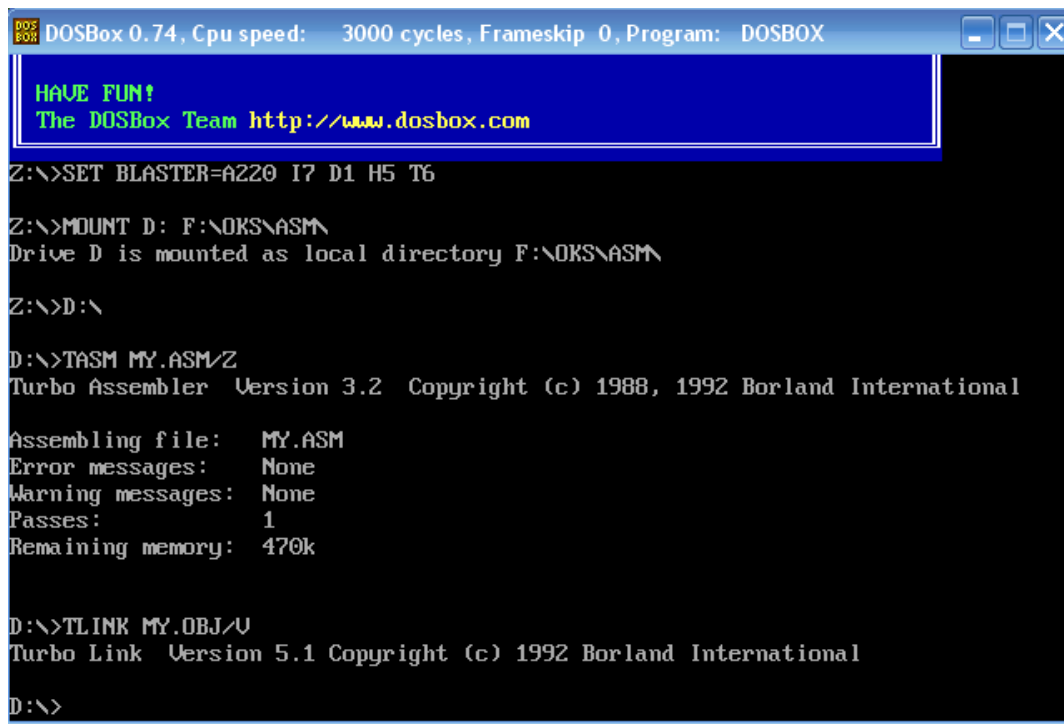
Результатом работы транслятора в случае отсутствия ошибок будет файл с расширением .OBJ – объектный модуль и тем же именем, что и исходный файл. В противном случае на экране появится перечень ошибок с указанием их типа и местоположения.

6. Странслированный без ошибок файл необходимо обработать компоновщиком (загрузчиком), т.е. набрать в командной строке следующую команду:

**TLINK \Путь\ Имя файла.OBJ /V**

На рис. 19 показан пример компоновки файла MY.OBJ.

Результатом при отсутствии ошибок будет файл с расширением *.EXE* – исполняемый файл, и именем, совпадающим с именем исходного файла. Эти файлы готовы к выполнению на компьютере. Их имена можно набрать на клавиатуре и нажать *ENTER*. Выполнение команд программы можно посмотреть в отладчике.



```
DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT D: F:\OKS\ASM\
Drive D is mounted as local directory F:\OKS\ASM\

Z:\>D:\

D:\>TASM MY.ASM\Z
Turbo Assembler Version 3.2 Copyright (c) 1988, 1992 Borland International

Assembling file: MY.ASM
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 470k

D:\>TLINK MY.OBJ\U
Turbo Link Version 5.1 Copyright (c) 1992 Borland International

D:\>
```

**Рис. 19.** Пример компоновки файла MY.OBJ

7. Для запуска отладчика необходимо набрать в командной строке команду:

**TD \Путь\Имя файла.EXE**

Данная команда приведёт к выводу на экран окна загрузчика Turbo Debugger (TD). Более подробно о работе с TD рассмотрим в следующем разделе.

### **3.2. Работа в отладчике Turbo Debugger**

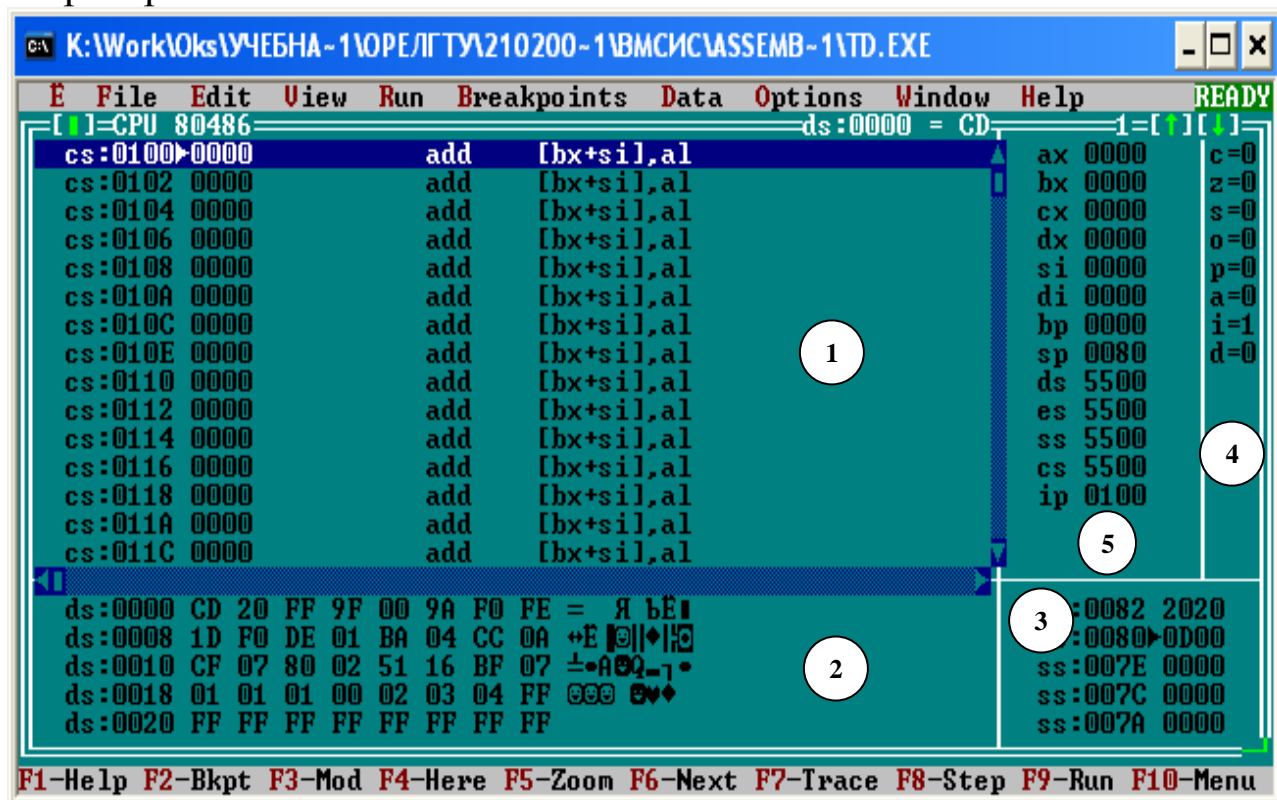
Отладчик TD позволяет по шагам проследить процесс выполнения программы на уровне регистров процессора и ячеек памяти. Внешний вид окна отладчика представлен на рис. 20.

Нижнее меню в отладчике - меню функциональных клавиш.

Значения некоторых функциональных клавиш:

- 1) **F7** – трассировка программы.
- 2) **F8** – выполнение программы по шагам т.е. по программе перемещается полоса выбора (синяя), и будет выполнена та команда, на которой эта полоса размещена.

**Примечание.** Трассировка по F7 отличается от пошагового выполнения по F8 тем, то при наличии подпрограмм при трассировке будет по шагам выполняться не только основная программа, но и каждая подпрограмма, которая вызывается из основной программы. А при пошаговом выполнении по F8 по шагам выполняется только основная программа, а каждая подпрограмма выполняется как единый оператор.



**Рис. 20.** Внешний вид окна TD

После выполнения команды на экране появляется содержимое регистров, флагов и адрес следующей на очереди команды (соответствующие регистры подсвечиваются белым цветом).

3) **F10** – выход в главное, верхнее меню.

Запускаются команды или с помощью мыши или с помощью клавиш перемещения курсора на клавиатуре. Курсором выбирается нужная команда и нажимается клавиша *ENTER* или нажимается левая кнопка мыши, если выбор выполнялся с помощью мыши. Выбор группы верхнего меню также может выполняться с помощью мыши или с клавиатуры (*ALT*+ горячая клавиши соответствующей группы).

В верхнем меню по команде *FILE* можно открыть любой файл, если он не был указан в команде *TD* при запуске отладчика.

По команде *VIEW* появляется еще меню, в котором находится команда *DUMP* – команда получения содержимого памяти по соответ-

ствующему адресу заданному в регистре *DS*, т.е. содержимое данных определенных в нашей программе. Данные начинаются с нулевого относительного адреса. Эти данные можно изменять.

*REGISTERS* – после запуска этой команды появляется окно с регистрами, и данные, находящиеся в этих регистрах, можно изменять. Курсором или мышкой выбрать изменяемый регистр и перевести курсор на изменяемое данное, на клавиатуре в появившемся окне набрать новое данное и нажать *ENTER*.

Выход из отладчика по нажатию *ALT+X*.

Выход из любой команды по нажатию клавиши *ESC*.

Заккрыть появившееся окно можно или нажать *ALT+F3* или надо перевести мышкой курсор в левый угол окна на зеленый квадрат и нажать левую кнопку мыши.

Верхнее и нижнее меню обрамляют отдельные окна, содержащие следующую информацию.

1. О сегменте кода текущей программы. В этом окне отражается смещение команды относительно начала сегмента кода (регистра *CS*), код команды, мнемоника команды и операнды в шестнадцатеричной системе счисления.

2. О сегменте данных текущей программы. В этом окне отражается смещение данных относительно начала сегмента (регистра *DS*), их представление в шестнадцатеричном виде и в виде символа таблицы *ASCII*.

3. О сегменте стека текущей программы. В этом окне отражается смещение вершины стека (регистра *SP*) относительно начала сегмента стека (регистра *SS*) и элементы стека в шестнадцатеричном виде.

4. О регистре флагов процессора. В этом окне перечислены все флаги (кроме *TF*) и в процессе выполнения команд программы отображается их реакция на выполненную команду.

5. О регистрах микропроцессора. В этом окне перечислены регистры процессора и в процессе выполнения команд программы отображается их содержимое после очередной выполненной команды.

Анализируя информацию из этих окон, можно делать вывод о правильности выполнения текущей программы.



## **4. ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

Лабораторные работы рассчитаны на студентов, не имеющих навыков программирования на языке ассемблера. Для выполнения лабораторных работ студенты должны уметь работать с графическим редактором, командной строкой операционной системы и иметь представление об электронных вычислительных машинах, системах счисления, а также принципах фон Неймана из курса информатики.

### **4.1. Общий порядок выполнения лабораторных работ**

При подготовке к лабораторной работе студенты должны повторить лекционный, а также дополнительный материал, относящийся к изучаемой теме, ознакомиться с теоретическими сведениями, а также ответить на контрольные вопросы, приведенные в соответствующих разделах настоящего практикума. В процессе изучения материала следует уделять особое внимание приведенным примерам и фрагментам программ.

Выполнение лабораторной работы заключается в выполнении соответствующего задания, состоящего из двух частей. В соответствии с первой частью задания, создаётся программа на языке ассемблера, исходный текст которой формируется в любом текстовом редакторе, например, блокноте. Затем после получения исполняемого файла осуществляется выполнение программы в отладчике TD (см. главу 3). При этом необходимо обращать особое внимание на моменты, изложенные во второй части задания, и делать соответствующие пометки в отчёте. Возникшие в ходе выполнения лабораторной работы вопросы решаются с преподавателем в течение аудиторного занятия.

По результатам выполнения лабораторной работы студенты дополняют отчет необходимыми сведениями (пояснениями, распечатками программ). В целом отчёт по лабораторной работе должен включать в себя:

1. Текст исходной программы на языке ассемблера с комментариями, в соответствии с первой частью задания.
2. Пояснения к вопросам и замечаниям, в соответствии со второй частью задания.
3. Краткие письменные ответы на контрольные вопросы лабораторной работы.

## **4.2. Лабораторная работа № 1. Линейное исполнение программ. Арифметические и поразрядные логические операции над целыми двоичными числами**

### **4.2.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при последовательном исполнении команд программы.
2. Приобретение навыков использования арифметических команд при написании ассемблерных программ.
3. Приобретение навыков использования поразрядных логических команд при написании ассемблерных программ.
4. Получение представления об особенностях обработки данных разных размерностей и режимах доступа к данным при выполнении арифметических и поразрядных логических операций.

### **4.2.2. Контрольные вопросы**

1. Понятие сегмента, характеристики сегмента, организация сегмента.
2. На какие сегменты разбита память компьютера? В какие регистры записываются начальные адреса сегментов?
3. Какие регистры процессора используются при выполнении арифметических операций?
4. На какие флаги воздействуют арифметические команды?
5. Особенности выполнения команд сложения и вычитания. Требования к операндам этих команд.
6. Особенности выполнения операции умножения. Особенности выполнения операции деления. Распределение регистров.
7. Основные логические операции и принципы их выполнения.
8. Правила формирования масок для установки и сброса битов.
9. Каким образом выполняются логические команды над словами?

### **4.2.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая выполняет арифметические и поразрядные логические операции над целыми двоичными числами.

1.1. В сегменте данных определить два байтовых значения 10 и 27 в десятичной системе счисления и однобайтовое произвольное число в двоичной системе счисления.

1.2. В сегменте данных зарезервировать байтовые ячейки для хранения суммы и разности с нулевыми первоначальными значениями, двухбайтовую ячейку для хранения произведения с единичным первоначальным значением, две байтовые ячейки для хранения остатка от деления и частного с произвольными первоначальными значениями.

1.3. Выполнить сложение 10 и 27; полученный результат записать в соответствующую ячейку памяти.

1.4. Выполнить вычитание 10 и 27; полученный результат переслать в соответствующую ячейку памяти.

1.5. Изменить знак второго числа (27) и снова выполнить операцию вычитания 10 и -27.

1.6. Выполнить умножение 10 и -27 с учетом знака; результат записать в соответствующую ячейку памяти. Выполнить умножение 10 и -27 без учета знака.

1.7. Выполнить деление 27 на 10; полученные результаты записать в соответствующие ячейки памяти.

1.8. Из сегмента данных в регистр переслать однобайтовое число в двоичной системе счисления, установить 2 любых бита в единицу, инвертировать все, сбросить 3 любых бита.

1.9. Полученный результат продублировать в другом регистре, сложить получившиеся значения по модулю два.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратит особое внимание на следующие моменты:

2.1. Как представляется число 27 и -27 в 16-ричной системе счисления?

2.2. Какое значение разности при выполнении вычитания в пунктах 1.4 и 1.5 задания и почему?

2.3. Где размещается результат умножения 10 и -27?

2.4. В чем заключается разность произведения 10 и -27 при умножении со знаком и без учета знака?

2.5. В каких регистрах размещаются результаты деления 27 и 10, и чему равны значения частного и остатка от деления?

2.6. Чему равна маска для установки двух битов в единицу и почему?

2.7. Чему равна маска для сброса трех битов в ноль?

### **4.3. Лабораторная работа № 2. Организация межсегментных переходов**

#### **4.3.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении межсегментных переходов.

2. Приобретение навыков использования команд сдвига при написании ассемблерных программ.

3. Получение представления об особенностях обработки данных и режимах доступа к данным при выполнении операций сдвига над данными.

#### **4.3.2. Контрольные вопросы**

1. Внутрисегментные и межсегментные переходы. Способы вычисления адресов переходов.

2. Разновидности внутрисегментных переходов и их особенности.

3. Флаги процессора и их использование в условиях.

4. Команды линейного логического и арифметического сдвигов. В чем заключается разница их выполнения? Области применения этих команд.

5. Особенности выполнения команд циклического сдвига. Сферы применения этих команд.

6. Что указывает директива *ASSUME* в программе?

#### **4.3.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая выполняет межсегментные переходы и операции сдвига над данными.

1.1. В программе определить один сегмент стека, три сегмента данных и три сегмента кода.

1.2. В сегменте стека зарезервировать 20 байт.

1.3. В первом сегменте данных определить однобайтовое число в двоичной системе счисления. Во втором сегменте данных определить адрес перехода на первый сегмент кода в виде двойного машинного слова. В третьем сегменте данных также определить однобайтовое число в двоичной системе счисления.

1.4. Начать выполнение с третьего сегмента кода и выполнить в нём с помощью команд линейного сдвига умножение на 2 числа из первого сегмента данных, а затем деление на 4 числа из третьего сегмента данных. Команды корректного завершения работы пометить меткой.

1.5. Затем перейти на метку, определенную во втором сегменте данных и выполнить переход в первый сегмент кода. Используя команды циклического сдвига, в регистре *BL* получить значение третьего бита числа из первого сегмента данных.

1.6. Далее выполнить переход во второй сегмент кода. В нём, используя команды циклического сдвига, в регистре *BH* получить значение пятого бита числа из третьего сегмента данных.

1.7. Затем перейти на метку конца, определенную в третьем сегменте кода и завершить работу программу.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратит особое внимание на следующие моменты:

2.1. Каким образом выполняется абсолютный прямой межсегментный переход, косвенный межсегментный переход?

2.2. В каких регистрах содержится адрес перехода при прямом переходе, при косвенном переходе?

2.3. В каких регистрах будут находиться результаты умножения на 2 и деления на 4 при выполнении операций линейного сдвига? Чему равны полученные результаты в десятичной системе счисления?

В чём преимущество использования команд сдвига для умножения и деления на степень двойки перед традиционными командами умножения и деления?

2.4. Чему равны третий и пятый биты анализируемых чисел, и какую позицию они занимают в регистрах *BL* и *BH*, соответственно?

2.5. Где находятся биты чисел, подвергнутые циклическому сдвигу, и чему они равны?

2.6. Какой из трёх сегментов кода является главным и почему?

#### **4.4. Лабораторная работа № 3. Команды условного и безусловного переходов. Организация ветвлений и циклов в программе**

##### **4.4.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении ветвлений и циклов.
2. Приобретение навыков использования команд условного и безусловного переходов, циклов при написании ассемблерных программ.
3. Получение представления об особенностях обработки данных, команд и режимах доступа к данным при организации ветвлений и циклов.

##### **4.4.2. Контрольные вопросы**

1. Ветвления в алгоритмах. Реализация ветвлений на языке ассемблера.
2. Команды условного и безусловного переходов. Каким образом вычисляются адреса переходов?
3. Циклы в алгоритмах. Организация циклов на языке ассемблера. Особенности и ограничения цикла *LOOP*.
4. В каком регистре находится во время выполнения программы смещение кода? Каким образом вычисляется адрес команды?
5. Какую принципиальную роль играет оператор безусловного перехода *JMP* при организации ветвлений?
6. Что означает корректное завершение программы?
7. Реальный и защищённый режимы работы процессора. Вычисление физических адресов ячеек памяти.

##### **4.4.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая реализует ветвления и циклы.
  - 1.1. В сегменте данных определить два числа в шестнадцатеричной системе счисления, размером в один байт каждое.
  - 1.2. Также в сегменте данных описать однобайтовую ячейку для хранения наибольшего общего делителя (НОД) двух чисел с произвольным первоначальным значением.
  - 1.3. Используя команды переходов и цикла, найти НОД двух чисел, описанных в сегменте данных.

1.4. Полученный результат поместить в соответствующую ячейку памяти.

1.5. Используя команды циклического сдвига, переходов и цикла подсчитать количество единиц в НОД.

1.6. Полученное значение поместить в регистр *DL*.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратит особое внимание на следующие моменты:

2.1. Как изменяется содержимое регистра *IP* при выполнении переходов? Какие виды переходов используются в программе. Что содержится в регистре *IP*?

2.2. Чему равен адрес следующей команды при выполнении условия для перехода и в противном случае?

2.3. Каким образом организованы циклы в программе?

2.4. Какое значение будет находиться в регистре для НОД после подсчета количества единиц? Сколько раз нужно выполнить команду циклического сдвига, чтобы получить первоначальное значение?

## **4.5. Лабораторная работа № 4. Обработка массивов. Числа Фибоначчи**

### **4.5.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций над массивами данных.

2. Приобретение навыков использования команд ассемблера, связанных с обработкой массивов.

3. получение представления об особенностях обработки данных, команд и режимах доступа к данным при обработке массивов.

### **4.5.2. Контрольные вопросы**

1. Массивы и их представление в памяти компьютера.

2. Режимы адресации данных, которые могут применяться для доступа к элементам массива. Приведите примеры.

3. Способы описания массивов в сегменте данных.

4. Особенности обработки двумерных массивов в ассемблерных программах. Вычисление смещения элемента двумерного массива относительно начала сегмента данных.

5. Какие режимы адресации данных можно использовать для доступа к элементам двумерного массива? Приведите примеры.

#### **4.5.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая использует массивы и позволяет вычислить числа Фибоначчи в заданном диапазоне.

1.1. В сегменте данных определить массив из 18 двухбайтовых ячеек с произвольным первоначальным значением, две двухбайтовые ячейки с нулевым первоначальным значением для размещения минимального и максимального элементов массива, соответственно.

1.2. Вычислить первые 18 чисел Фибоначчи и поместить их в массив, обращаясь к нему как к одномерному массиву.

1.3. Рассматривая имеющийся массив как двумерный размера  $3 \times 6$  (3 строки, 6 столбцов), найти наименьших из нечётных элементов второй строки и наибольший из чётных элементов четвёртого столбца.

1.4. Полученные результаты поместить в соответствующие ячейки памяти.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратит особое внимание на следующие моменты:

2.1. Как расположены элементы массива в памяти?

2.2. Сколько ячеек памяти отведено под массив?

2.3. Каким образом осуществляется доступ к элементам одномерного массива? Как изменяется индексный регистр, используемый для указания смещения элемента массива?

2.4. В каких регистрах процессора содержатся смещения по строке и смещение по столбцу для двумерного массива? Каким образом они вычисляются? На что указывает метка начала массива?

### **4.6. Лабораторная работа № 5. Использование подпрограмм.**

#### **Сортировка массива чисел**

##### **4.6.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении переходов, связанных с вызовами подпрограмм.



2. Приобретение навыков использования команд безусловного перехода для обработки процедур при написании ассемблерных программ.

3. Получение представления об особенностях обработки данных, команд и режимах доступа к данным при организации вызовов процедур.

#### **4.6.2. Контрольные вопросы**

1. Описание процедур. Варианты размещения процедур в программе.

2. Процедуры и сопрограммы. Особенности передачи управления при вызове процедур и при вызове сопрограмм.

3. Команды вызова процедуры и возврата из неё.

4. Механизмы обработки процедур ближнего и дальнего вызовов. Что представляет собой «адрес возврата» и где он размещается?

5. Обязательно ли наличие сегмента стека в программе, содержащей процедуры, и почему?

6. Сопрограммы. Принципы взаимодействия сопрограмм.

#### **4.6.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая выполняет сортировку массива чисел по возрастанию, используя процедуры.

1.1. В сегменте стека зарезервировать 5 двухбайтовых ячеек.

1.2. В сегменте данных определить два массива из 8 однобайтовых ячеек. В первом массиве разместить исходные значения, во втором массиве - нули.

1.3. Используя один из алгоритмов сортировки, выполнить упорядочивание элементов массива по возрастанию.

1.4. Фрагмент программы, соответствующий сортировке, представить в виде процедуры ближнего вызова. Вариант размещения процедуры в программе выбрать самостоятельно.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. Как расположены процедуры в сегменте кода?

2.2. Что содержит регистр *IP* при выполнении команды *CALL*?

2.3. Каким образом изменяется состояние стека при обращении к процедуре ближнего вызова?

2.4. Что представляет собой адрес возврата и чему он равен? Как изменяется состояние стека при возврате из процедуры?

2.5. В какую точку основной программы выполняется возврат из процедуры?

2.6. Удалить из программы сегмент стека. Запустить программу на исполнение. Объяснить, что происходит при вызове процедуры сортировки.

#### **4.7. Лабораторная работа № 6. Обработка структур. Ведение базы данных о пациентах**

##### **4.7.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций над структурами.
2. Приобретение навыков использования команд для работы со структурами на примере обработки информации о пациентах.
3. Получение представления об особенностях обработки данных, команд и режимах доступа к данным при обработке структур.

##### **4.7.2. Контрольные вопросы**

1. Структуры и определение шаблона структуры в программе.
2. Инициализация полей структуры в программе?
3. Режимы адресации для доступа к элементам структуры, для доступа к элементам массива структур.
4. Каким образом вычисляется расстояние до некоторого поля отдельного элемента массива структур?
5. Назначение оператора TYPE.

##### **4.7.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая использует структуры для хранения и обработки информации о пациентах.

1.1. Определить шаблон структуры для хранения информации о пациентах, состоящей из следующих полей:

номер медкарты – dw с нулевым начальным значением;  
пол – db с произвольным начальным значением;  
год рождения – dw с нулевым начальным значением;  
дата поступления – db с шаблоном ‘/ /’;  
дата выписки – db с шаблоном ‘/ /’.

1.2. В сегменте данных определить три экземпляра записи, указав конкретную информацию о трёх пациентах.

1.3. В сегменте кода вывести информацию о количестве пациентов, поступивших на конкретную дату.

1.4. Получить сведения о количестве пациентов женского пола, которые были выписаны на определённую дату.

1.5. Найти год рождения пациента по номеру медкарты.

1.6. Найти количество пациентов мужского пола по указанному году рождения.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратить особое внимание на следующие моменты:

2.1. В каком месте программы расположено описание шаблона структуры?

2.2. Как расположены элементы структуры в памяти, массива структур в памяти?

2.3. Сколько байтов отведено под одну структуру, под весь массив структур?

2.4. Каким образом осуществляется доступ к элементам массива структур, к полям отдельной структуры? Как изменяются значения базовых и индексных регистров?

2.5. На что указывает метка начала массива? На что указывает название поля структуры?

## **4.8. Лабораторная работа № 7. Использование стека. Проверка баланса расстановки скобок в строке**

### **4.8.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций со стеком и строками.

2. Приобретение навыков использования команд для работы со стеком и строками.

3. Получение представления об особенностях обработки данных, команд и режимах доступа к данным при использовании стека.

### **4.8.2. Контрольные вопросы**

1. Память с последовательным доступом. Виды памяти с последовательным доступом.

2. Определение стека. Организация стека.

3. Команды работы со стеком.

4. Какие регистры используются при работе со стеком? Каково их назначение.

5. Как изменяется содержимое указателя вершины стека при включении нового элемента в стек и извлечении элемента из стека? Почему?

#### **4.8.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая использует стек для проверки баланса расстановки скобок в строке символов.

1.1. В сегменте стека определить стек из 20 двухбайтовых ячеек с начальным значением в виде символа «\$».

1.2. В сегменте данных определить строку не более 20 символов, содержащую произвольное арифметическое выражение, в котором используются три вида скобок «( )», «[ ]» и «{ }». Последовательность и вложенность скобок может быть любая.

1.3. Также в сегменте данных определить байтовую ячейку для сохранения результата проверки.

1.4. В сегменте кода, используя стек, проверить, все ли скобки закрыты, соответствует ли каждая закрывающаяся скобка открывающейся и нет ли лишних закрывающихся скобок.

1.5. В соответствующую ячейку памяти поместить код результата проверки (0 – скобки расставлены правильно, 1 - несоответствие скобок, 2 – не все скобки закрыты, 3 – лишние закрывающиеся скобки).

1.6. Если возникла ошибка несоответствия скобок, то в регистр *DL* поместить код скобки, которая ожидается, иначе – 0.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратит особое внимание на следующие моменты:

2.1. С какого адреса в памяти начинается сегмент стека? Какое значение содержится в регистре *SP* перед началом загрузки элементов в стек? На что указывает регистр *SP*?

2.2. Как изменяется содержимое регистра *SP* при помещении одного элемента в стек, при извлечении одного элемента из стека?

2.3. В каком порядке помещаются в стек и извлекаются элементы из стека?

2.4. Что является признаком опустошения стека?

## **4.9. Лабораторная работа № 8. Использование стека и рекурсивных процедур. Организация передачи параметров через стек в процедуру вычисления факториала числа**

### **4.9.1. Цель работы**

Цели лабораторной работы:

1. Изучение принципов функционирования памяти и микропроцессора компьютера при выполнении операций со стеком для передачи параметров через стек и рекурсивными подпрограммами.
2. Приобретение навыков использования команд для работы со стеком и подпрограммами для организации передачи параметров через стек.
3. Получение представления об особенностях обработки данных, команд и режимах доступа к данным при организации передачи параметров в подпрограммы через стек.

### **4.9.2. Контрольные вопросы**

1. Какие регистры используются при работе со стеком? Назначение регистра *BP*.
2. Когда передаваемые в процедуру аргументы записываются в стек? Какое место они занимают в стеке после входа в процедуру?
3. Формат процедуры при использовании передачи параметров через стек. Пролог и эпилог процедуры.
4. Какие действия выполняются в вызывающей программе после возврата из процедуры и для чего?
5. Как вычисляется адрес требуемого параметра в стеке в процедурах ближнего вызова? Как вычислить адрес аргумента в процедуре дальнего вызова?
6. Передача параметров по ссылке и ее особенности. Какой оператор используется для записи в регистр адреса данного, а не самого данного?
7. Передача параметров по значению и ее особенности

### **4.9.3. Задание на лабораторную работу**

1. Написать программу на языке ассемблера, которая использует стек для передачи параметров в процедуру вычисления факториала числа.

1.1. В сегменте данных определить две двухбайтовые ячейки с одинаковым значением:  $3h$ , ниже еще две двухбайтовые ячейки для сохранения результата с произвольным первоначальным значением.

1.2. В сегменте кода описать процедуру ближнего вызова, в которой содержится программа вычисления факториала числа.

1.3. Число, для которого необходимо вычислить факториал должно передаваться в качестве аргумента через стек.

1.4. В сегменте кода в основной программе выполнить дважды вызов процедуры вычисления факториала: в первом случае аргумент передается по значению (из первой ячейки), во втором случае – по ссылке (из второй ячейки).

1.5. Результаты вычисления факториала записать в соответствующие ячейки памяти, определенные в сегменте данных.

2. На основе исходной программы получить исполняемый файл. Выполнить программу по шагам с помощью отладчика TD, описать изменение состояния регистров и ячеек памяти при выполнении программы. Обратит особое внимание на следующие моменты:

2.1. Каково содержимое стека до входа в процедуру и после?

2.2. Как изменяется содержимое стека при рекурсивных вызовах процедуры и возвратах из рекурсии?

2.3. На что указывает регистр SP после выполнения первой команды эпилога процедуры?

2.4. Каким образом в программе осуществляется передача параметра по значению и по ссылке?

2.5. Какое значение имеют ячейки памяти, отведенные под исходные данные, и почему?

## **5. ЗАДАНИЯ К ПРАКТИЧЕСКИМ ЗАНЯТИЯМ**

Практические задания рассчитаны на студентов, имеющих некоторые навыки программирования на языке ассемблера, полученные ими в ходе выполнения лабораторных, идущих параллельно с практическими занятиями. Для выполнения практических заданий студенты должны уметь работать с текстовым редактором, командной строкой операционной системы и иметь представление об электронных вычислительных машинах, системах счисления, а также принципах фон Неймана из курса информатики.

### **5.1. Общий порядок выполнения практических заданий**

При подготовке к практическому занятию студенты должны повторить лекционный и дополнительный материал, относящийся к изучаемой теме, ознакомиться с теоретическими сведениями, а также ответить на контрольные вопросы, приведенные в соответствующих разделах настоящего практикума. В процессе изучения материала следует уделять особое внимание приведенным примерам и фрагментам программ.

Подготовка к практическому занятию является самостоятельной работой студентов и выполняется студентами до аудиторного занятия. Возникшие вопросы решаются совместно с преподавателем в начале аудиторного занятия.

В ходе практического занятия студенты выполняют соответствующие задания. В соответствии с заданием, создаётся программа на языке ассемблера, исходный текст которой формируется в любом текстовом редакторе, например, блокноте. Затем после получения исполняемого файла осуществляется выполнение программы на виртуальной машине DOSBox (см. главу 3). Возникшие при выполнении задания вопросы решаются с преподавателем в течение аудиторного занятия.

По результатам выполнения практического задания студенты дополняют отчёт необходимыми сведениями (пояснениями, распечатками программ). В целом отчёт по практическому занятию должен включать в себя:

1. Текст исходной программы на языке ассемблера с комментариями, в соответствии с заданием.
2. Краткие письменные ответы на контрольные вопросы практического занятия.

## **5.2. Практическое занятие № 1. Управление дисплеем. Вывод символов ASCII на экран**

### **5.2.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов работы дисплея и функций ОС и BIOS для вывода отдельных символов с атрибутами в текстовом режиме.
2. Изучение принципов работы системных часов.
3. Приобретение навыков использования функций ОС и BIOS для управления дисплеем для вывода отдельных символов с атрибутами в текстовом режиме.
4. Приобретение навыков использования функций управления системными часами для организации задержек в ассемблерных программах.

### **5.2.2. Контрольные вопросы**

1. Основные части видеосистемы. Понятие дисплейной страницы.
2. Сколько байтов отводится на каждую позицию экрана в текстовом режиме? Какое разрешение имеет дисплей в текстовом режиме?
3. Функции управления курсором в текстовом режиме.
4. Функции вывода на экран одного символа. Особенности вывода чисел на экран
5. Распределение битов в байте атрибутов
6. Функции очистки экрана.
7. Организация задержки с помощью системных часов.

### **5.2.3. Практическое задание**

Написать программу на языке ассемблера, которая выводит все символы таблицы кодов *ASCII* на экран в указанную позицию курсора. Символы выводятся в одну и ту же позицию с заданной задержкой и заданным цветом. Перед выводом символов экран необходимо очистить. Позиция на экране, время задержки и варианты распределения цвета символов выбираются самостоятельно.

## **5.3. Практическое занятие № 2. Управление дисплеем в графическом режиме**

### **5.3.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов видеосистемы в графическом режиме.



2. Приобретение навыков использования функций, необходимых для работы в графическом режиме при написании ассемблерных программ.

### **5.3.2. Контрольные вопросы**

1. Особенности растровой и векторной графики.
2. Изменение цвета фона в графическом режиме.
3. Типы видеоадаптеров.
4. Функции рисования точки и чтения цвета точки в графическом режиме.
5. Особенности вывода текста в графическом режиме.

### **5.3.3. Практическое задание**

Написать программу на языке ассемблера, которая в графическом режиме рисует прямоугольник размера 100×300 пикселей в указанной позиции экрана и закрашивает этот прямоугольник заданным цветом. Затем моделирует движение отрезка длиной 5 пикселей от левой границы прямоугольника до правой границы. При достижении правой границы отрезок останавливается.

## **5.4. Практическое занятие № 3. Управление клавиатурой. Проверка символа в буфере клавиатуры**

### **5.4.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов организации и функционирования буфера клавиатуры.
2. Приобретение навыков использования функций ОС и BIOS для считывания отдельных символов с клавиатуры.
3. Получение представления о процессе обработки нажатий отдельных клавиш и комбинаций клавиш клавиатуры.

### **5.4.2. Контрольные вопросы**

1. Принципы организации буфера клавиатуры.
2. Понятие скан-кода, ASCII-кода, расширенного кода.
3. Процесс обработки нажатия клавиши на клавиатуре.
4. Функции ОС и BIOS для очистки буфера и проверки буфера клавиатуры.
5. Функции ОС и BIOS для ввода символов с клавиатуры. Особенности ввода символов с эхом, ожиданием, без эха, без ожидания.

### **5.4.3. Практическое задание**

Написать программу на языке ассемблера, которая обрабатывает нажатия клавиш на клавиатуре и выдаёт на экран следующую информацию:

1. *ASCII*-код символа, если была нажата алфавитно-цифровая клавиша основной клавиатуры.
2. Строку «Расширенный код», если была нажата функциональная клавиша, клавиши дополнительной клавиатуры или комбинации клавиш с «*ALT*» и «*CTRL*».

Программа должна предварительно чистить буфер клавиатуры. Информация о каждом нажатии должна выводиться в новой строке (предыдущие сведения не затираются). Символы должны вводиться без эха. Программа должна завершиться по нажатию клавиши «*ESC*».

## **5.5. Практическое занятие № 4. Управление клавиатурой. Ввод строки символов**

### **5.5.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов организации структур данных для хранения строк.
2. Приобретение навыков использования функций ОС и BIOS, связанных с вводом строк символов с клавиатуры и выводом их на экран.

### **5.5.2. Контрольные вопросы**

1. Структуры данных для хранения строк. Особенности хранения строк, введённых с клавиатуры.
2. Функция ОС для ввода строки символов с клавиатуры. Особенности её работы.
3. Функция ОС для вывода строки символов на экран. Особенности её работы.

### **5.5.3. Практическое задание**

Написать программу на языке ассемблера, которая позволяет вводить с клавиатуры и выводить на экран строки символов. Строки должны храниться в одном и том же буфере. Группы строк (вводимая и выводимая) должны отделяться друг от друга строкой символов, например «\*». Программа должна завершать свою работу при нажа-

тии клавиш «*CTRL*» + «*C*». Максимальная длина строки выбирается самостоятельно.

## **5.6. Практическое занятие № 5. Файлы последовательного доступа. Запись и чтение информации**

### **5.6.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов организации каталогов и файлов последовательного доступа.
2. Приобретение навыков использования функций ОС для работы с каталогами и файлами последовательного доступа.

### **5.6.2. Контрольные вопросы**

1. Файлы последовательного доступа. Запись в файл.
2. Файлы последовательного доступа. Чтение из файла.
3. Позиционирование в файле.
4. Что является признаком конца файла?
5. Функции ОС для создания и удаления файлов.

### **5.6.3. Практическое задание**

Написать программу на языке ассемблера, которая создаёт подкаталог, в нём – файл; помещает в файл две строки, введённые с клавиатуры, а затем выводит их на экран. Путь к создаваемому каталогу и файлу, а также название каталога и файла выбирается самостоятельно. Файл рассматривается как файл последовательного доступа, вводимые строки могут иметь различную длину.

## **5.7. Практическое занятие № 6. Файлы прямого доступа. Запись и чтение информации**

### **5.7.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов организации каталогов и файлов прямого доступа.
2. Приобретение навыков использования функций ОС для работы с каталогами и файлами прямого доступа.

### **5.7.2. Контрольные вопросы**

1. Файлы прямого доступа. Запись в файл.
2. Файлы прямого доступа. Чтение из файла.

3. Функции ОС для открытия и закрытия файлов.
4. Каким образом вычисляется расстояние (смещение) до некоторой записи файла прямого доступа?

### ***5.7.3. Практическое задание***

Написать программу на языке ассемблера, которая создаёт подкаталог, в нём – файл; помещает в файл три строки, введённые с клавиатуры, а затем выводит вторую строку на экран. Путь к создаваемому каталогу и файлу, а также название каталога и файла выбирается самостоятельно. Файл рассматривается как файл прямого доступа. Вводимые строки могут иметь различную длину, максимальная длина записи 10 символов; недостающие до максимальной длины символы введённой строки заполняются пробелами.

## **5.8. Практическое занятие № 7. Управление дисками.**

### **Организация поиска каталогов и файлов**

#### ***5.8.1. Цель практического занятия***

Цели практического занятия:

1. Изучение принципов организации дисков, каталогов, файлов.
2. Приобретение навыков использования функций BIOS и ОС для управления дисками, каталогами, файлами в ассемблерных программах.

#### ***5.8.2. Контрольные вопросы***

1. Функции ОС для определения свободного пространства на диске.
2. Функции ОС для чтения и изменения атрибутов файла. Какие атрибуты имеет файл?
3. Функции ОС для организации поиска файлов.
4. Что такое временный файл? Функции ОС для работы с временными файлами.

#### ***5.8.3. Практическое задание***

Написать программу на языке ассемблера, которая проверяет, какой дисковый накопитель установлен по умолчанию, создаёт на диске каталог, а в нём 5 файлов: два файла имеют установленный атрибут «Только для чтения», два – «Скрытый», один файл – обычный. После этого программа осуществляет поиск файлов в каталоге по имени и расширению, используя маски.

При наличии соответствующих файлов выводится информация о них на экран, в противном случае – сообщение об отсутствии иско-  
мых файлов. Имена файлов, каталога, расширения файлов, маски для  
поиска выбираются самостоятельно.

## **5.9. Практическое занятие № 8. Управление мышью**

### **5.9.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов организации и функционирования мани-  
пулятора «мышь» персонального компьютера.
2. Приобретение навыков использования функций прерывания  
33h для управления мышью в ассемблерных программах.

### **5.9.2. Контрольные вопросы**

1. Для каких целей используется манипулятор «мышь» в персо-  
нальном компьютере? Виды манипуляторов «мышь».
2. Что такое микки? Как рассчитывается положение курсора мы-  
ши в различных режимах экрана?
3. Функции для установки драйвера мыши и проверки её состоя-  
ния.
4. Что представляет собой обработчик событий мыши? Функция  
для управления обработчиками событий мыши и её особенности.

### **5.9.3. Практическое задание**

Написать программу на языке ассемблера, которая проверяет  
подключение мыши к персональному компьютеру. Если мышь под-  
ключена, выводится соответствующее сообщение и выполняется об-  
работка двух событий мыши. Если мышь не подключена, программа  
завершает работу с выводом соответствующего сообщения на экран.

Обработка первого события заключается в том, что при нажатии  
левой кнопки мыши в соответствующей позиции курсора в текстовом  
режиме должен печататься символ (выбирается самостоятельно). Об-  
работка второго события сводится к тому, что при нажатии правой  
кнопки на ранее напечатанной на экране строке о наличии мыши,  
программа должна завершить работу.

## **5.10. Практическое занятие № 9. Управление прерываниями. Написание собственного прерывания**

### **5.10.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов организации пользовательского прерывания в ассемблерных программах.
2. Приобретение навыков использования функций ОС для написания пользовательского прерывания в ассемблерных программах.

### **5.10.2. Контрольные вопросы**

1. Организация пользовательского прерывания в ассемблерных программах.
2. Ловушки, которые могут возникнуть при написании пользовательского прерывания.
3. Функции ОС для изменения и восстановления векторов прерываний. Особенности их работы.
4. Последовательность действий при обработке прерывания.
5. Исключения. Виды исключений. Обработка исключений.

### **5.10.3. Практическое задание**

Написать программу на языке ассемблера, которая обращается к пользовательскому прерыванию. Эффект прерывания выбрать самостоятельно.

## **5.11. Практическое занятие № 10. Управление счётчиком времени суток**

### **5.11.1. Цель практического занятия**

Цели практического занятия:

1. Изучение принципов функционирования счётчика времени суток персонального компьютера.
2. Приобретение навыков использования функций ОС для доступа к счётчику времени суток при работе с датой и временем в ассемблерных программах.

### **5.11.2. Контрольные вопросы**

1. Каналы микросхемы таймера. Канал для доступа к счётчику времени суток.
2. Функции ОС для чтения установки даты. Особенности их работы.

3. Функции ОС для чтения установки времени. Особенности их работы.

4. Какое количество импульсов счётчика времени суток эквивалентно 10 секундам, 1 минуте, полутора (1,5) часам?

### ***5.11.3. Практическое задание***

Написать программу на языке ассемблера, которая выдаёт на экран значение текущей даты, затем пытается изменить дату на новую и, в случае успеха выводит на экран новую дату, в противном случае – сообщение о невозможности изменить текущую дату.

## **5.12. Практическое занятие № 11. Генерация звука**

### ***5.11.1. Цель практического занятия***

Цели практического занятия:

1. Изучение принципов функционирования динамика и таймера персонального компьютера для формирования звуковых сигналов.

2. Приобретение навыков использования функций ОС для управления динамиком и таймером персонального компьютера при генерации звуковых сигналов в ассемблерных программах.

### ***5.11.2. Контрольные вопросы***

1. Микросхема таймера. Каналы микросхемы таймера. Канал для управления звуком.

2. Процесс генерации звука. Управление взаимодействием таймера и динамика для генерации звуковых сигналов.

3. Команды отключения и включения аппаратных прерываний. Для чего отключаются аппаратные прерывания в процессе генерации звука? Влияние отключения аппаратных прерываний на счётчик времени суток.

4. Какой порт используется для изменения частоты сигнала при генерации звука?

### ***5.11.3. Практическое задание***

Написать программу на языке ассемблера, которая выдаёт через динамик гамму нот (от ноты ДО до ноты СИ). Ноты выводятся друг за другом через определённый интервал времени (например, 3 секунды).

## ЛИТЕРАТУРА

1. DOSBox [Электронный ресурс]. – Режим доступа: <https://ru.wikipedia.org/wiki/DOSBox>. – Систем. требования: P IV; 64 Мб ОЗУ; Windows 98 и выше; SVGA 32768 и более цветов; 640×480; мышь; IE 4.0 и выше. – Загл. с экрана.
2. Абель, П. Язык ассемблера для IBM PC и программирования [Текст]/П. Абель/ Пер. с англ. Ю.В. Сальникова.- М.: Высшая школа, 1992.-447 с., ил.
3. Аппаратные средства вычислительной техники : учебник для вузов/ В.А. Минаев, А.П. Фисун, В.А. Зернов, В.Т. Еременко, И.С. Константинов, А.В. Коськин, Ю.А. Белевская, С.В. Дворянкин. – Орел: Изд-во ОрелГТУ; Орел: Изд-во ОГУ, 2010. – 461 с. (Информационная безопасность социотехнических систем)
4. Архитектура ЭВМ. Вычислительные системы, сети и телекоммуникации [Электронный ресурс]. – Режим доступа: <http://rstud.ru>. – Систем. требования: P IV; 64 Мб ОЗУ; Windows 98 и выше; SVGA 32768 и более цветов; 640×480; мышь; IE 4.0 и выше. – Загл. с экрана.
5. Бройдо, В.Л. Вычислительные системы, сети и телекоммуникации: учебное пособие для вузов/ В.Л. Бройдо. – 2-е изд. – СПб.: Питер, 2005. – 703 с.; ил.
6. Костенко, Т.П. Организация ЭВМ и систем: учебное пособие для вузов/ Т.П. Костенко. – Орел : Изд-во ОрелГТУ , 2006. – 152 с.
7. Мелехин, В.Ф. Вычислительные машины, системы и сети: учебник для вузов/ В.Ф. Мелехин, Е.Г. Павловский. – 2-е изд. – М.: Издательский центр «Академия», 2007. – 560 с.
8. Пирогов, П.Ю. ASSEMBLER. Учебный курс [Текст]/ П.Ю. Пирогов.- М.: Издатель Молгачева С.В.- Нолидж, 2001. – 848 с. – ил. – ISBN: 5-89251-101-4
9. Ремонтов, А.П. Вычислительные машины и системы: учебное пособие [Электронный ресурс]/ А.П. Ремонтов, А.А. Писарев. – Режим доступа: [http://window.edu.ru/window\\_catalog/files/r53969/stup323.pdf](http://window.edu.ru/window_catalog/files/r53969/stup323.pdf). – Систем. требования: P IV; 64 Мб ОЗУ; Windows 98 и выше; SVGA 32768 и более цветов; 640×480; мышь; IE 4.0 и выше. – Загл. с экрана.
10. Таненбаум, Э. Архитектура компьютера/ Э. Таненбаум. – 4-е изд. – СПб.: Питер, 2003. – 698 с.; ил.
11. Цилькер, Б.Я. Организация ЭВМ и систем: учебник для вузов/ Б.Я. Цилькер, С.А. Орлов. – СПб.: Питер, 2007. – 668 с.; ил.



12. Юров, В.И. Assembler [Текст]/ В.И. Юров.- Учебник для вузов.- 2-е издание.- СПб.: Питер, 2006.- 637 с.: ил.- ISBN: 5-94723-581-1

13. Юров, В.И. Assembler. Практика [Текст]/ В.И. Юров.- Учебник для вузов.- 2-е издание.- СПб.- Питер, 2006.- 399 с.: ил.- ISBN: 5-94723-671-0

Учебное издание

*Конюхова Оксана Владимировна  
Кравцова Эльвира Александровна*

**ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ МАШИН И  
СИСТЕМ. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АСЕМБЛЕРА**

Практикум

Редактор

Технический редактор

Федеральное государственное бюджетное образовательное  
учреждение высшего профессионального образования  
«Государственный университет-учебно-научно-производственный комплекс»  
Лицензия ИД № 00670 от 05.01.2000 г.

Отпечатано с готового оригинал-макета на полиграфической базе ФГБОУ ВПО  
«Госуниверситет-УНПК»

Подписано к печати на полиграфической базе ФГБОУ ВПО «Госуниверситет -  
УНПК», 302030, г. Орел, ул. Московская, 65. Формат 60x84 1/16

Усл.печ.л. \_\_\_\_\_ . Тираж \_\_\_\_ экз.

Заказ № \_\_\_\_\_