ОБЩИЕ УКАЗАНИЯ
к выполнению лабораторных работ по курсу
"Системное программирование"

ЛАБОРАТОРНАЯ РАБОТА №1 - 2

ЗНАКОМСТВО С АССЕМБЛЕРОМ.

Цель работы: Знакомство с процессом компиляции ассемблерной программы, процессом ее получения и основными возможностями.

1. Краткие теоретические сведения
1.1. Структура ассемблерной программы

Каждый язык программирования имеет свои особенности. Язык ассемблера - не исключение. Традиционно первая программа выводит приветственное сообщение на экран ‘Hello World’ .

В отличие от многих современных языков программирования в ассемблерной программе каждая команда располагается на ОТДЕЛЬНОЙ СТРОКЕ. Нельзя разместить несколько команд на одной строке. Не принято, также, разбивать одну команду на несколько строк.

Язык ассемблера является РЕГИСТРОНЕЧУВСТВИТЕЛЬНЫМ. Т.е. в большинстве случаев нет разницы между большими и малыми буквами. Команда может быть ДИРЕКТИВОЙ - указанием транслятору. Они выполняются в процессе превращения программы в машинный код. Многие директивы начинаются с точки. Для удобства чтения программы они обычно пишутся БОЛЬШИМИ БУКВАМИ. Кроме директив еще бывают ИНСТРУКЦИИ - команды процессору. Именно они и будут составлять машинный код программы.

Нужно отметить, что понятие "машинного кода" очень условно. Часто оно обозначает просто содержимое выполняемого файла, хранящего кроме собственно машинных команд еще и данные. В нашем случае это будет текст выводимого сообщения "Hello".

1.2. Особенности создания ассемблерной программы в среде DOS средствами TASM и MASM

Язык ассемблера является самым низкоуровневым языком программирования. Т.е. он ближе любых других приближен к архитектуре ЭВМ и ее аппаратным возможностям, позволяя получить к ним полный доступ. В отличие от языков высокого уровня (ЯВУ) ассемблерная программа содержит только тот код, который ВВЕЛ ПРОГРАММИСТ. Никаких дополнительных "обвязок". Вся ответственность за "логичность" кода ПОЛНОСТЬЮ лежит на узких плечах ПРОГРАММИСТА.

Простой пример. Обычно подпрограммы заканчиваются командой возврата. Если ее не задать явно, транслятор все равно добавит ее в конец подпрограммы. Ассемблерная подпрограмма без команды возврата НЕ ВЕРНЕТСЯ в точку вызова, а будет выполнять код, следующий за подпрограммой, как-будто он является ее продолжением. Еще пример. Можно попробовать "выполнить" данные вместо кода. Часто это лишено смысла. Но если программист это сделает, транслятор промолчит. Язык ассемблера позволяет делать все! Тут нет НИКАКИХ ограничений. Но с другой стороны это часто является источником ошибок.

Эти особенности приводят к тому, что ассемблерные программы часто "подвешивают" компьютер, особенно у начинающих программистов.

Выделим три разновидности "зависания" по способу борьбы с ним.

* Простое - для выхода из него достаточно нажать Ctrl+Break или Ctrl+C (сначала нажимается клавиша Ctrl и, НЕ ОТПУСКАЯ ее, нажимается вторая клавиша - C или Break;

отпускаются в обратном порядке). Программа при этом аварийно завершается выходом в DOS.

* Мягкое - машина не реагирует на Ctrl+Break, но клавиатура "дышит". Т.е. при нажатии на клавиши, типа NumLock, моргают соответствующие светодиоды. В этом случае машину нужно будет перегрузить, нажав Ctrl+Alt+Del. В среде Windows нужно просто "убить" сеанс, закрыв окно.

Жесткое - машина никак не реагирует на клавиатуру и не воспринимает комбинацию Ctrl+Alt+Del. В этом случае поможет аппаратный сброс при помощи кнопки "Reset", расположенной на передней панели системного блока. Не нужно ВЫКЛЮЧАТЬ и включать ЭВМ. Вы как будущие разработчики аппаратуры должны знать, что она выходит из строя в основном при включении и выключении.

1.3. Процесс обработки программы на языке ассемблера

Из-за своей специфики, а также по традиции, для программирования на языке ассемблера нет никаких сред-оболочек типа Turbo C, Turbo Pascal и т.д. Тут приходится пользоваться "утилитами командных строк", как 30 лет назад. Весь процесс технического создания ассемблерной программы можно разбить на 4 шага (исключены этапы создания алгоритма, выбора структур данных и т.д.).

* Набор программы в текстовом редакторе и сохранение ее в отдельном файле. Каждый файл имеет имя и тип, называемый иногда расширением. Тип в основном используется для определения назначения файла. Например, программа на C имеет тип C, на Pascal - PAS, на языке ассемблера - ASM.

* Обработка текста программы транслятором. На этом этапе текст превращается в машинный код, называемый объектным. Кроме того, есть возможность получить листинг программы, содержащий кроме текста программы различную дополнительную информацию и таблицы, созданные транслятором. Тип объектного файла - OBJ, файла листинга - LST. Этот этап называется ТРАНСЛЯЦИЕЙ.

* Обработка полученного объектного кода компоновщиком. Тут программа "привязывается" к конкретным условиям выполнения на ЭВМ. Полученный машинный код называется выполняемым. Кроме того, обычно получается карта загрузки программы в ОЗУ. Выполняемый файл имеет тип EXE, карта загрузки - MAP. Этот этап называется КОМПОНОВКОЙ или ЛИНКОВКОЙ.

* Запуск программы. Если программа работает не совсем корректно, перед этим может присутствовать этап ОТЛАДКИ программы при помощи специальной программы - отладчика. При нахождении ошибки приходится проводить коррекцию программы, возвращаясь к шагу 1. Таким образом, процесс создания ассемблерной программы можно изобразить в виде следующей схемы. Конечной целью, напомним, является работоспособный выполняемый файл HELLO.EXE.

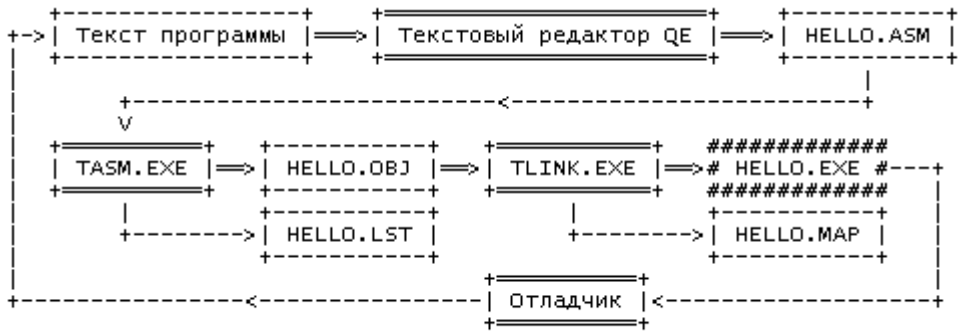

Рис.1

1.4. Особенности создания ассемблерной программы в среде эмулятора EMU8086

Этот программный продукт содержит все необходимое для создания программы на языке Assembler.

Пакет Emu8086 сочетает в себе продвинутый текстовый редактор, assembler, disassembler, эмулятор программного обеспечения (Виртуальную машину) с пошаговым отладчиком, примеры.

В процессе выполнения программы мы можем наблюдать программные регистры, флаги и память, АЛУ показывает работу центрального процессора.

Встроенная виртуальная машина полностью блокирует обращение программы к реальным аппаратным средствам ЭВМ, накопителям памяти, это делает процесс отладки намного более легкой

1.4. Правила оформления ассемблерных программ

При наборе программ на языке ассемблера придерживайтесь следующих правил:
* директивы набирайте большими буквами, инструкции - малыми;
* пишите текст широко - не скупердяйничайте;
* не выходите за край экрана, т.е. не делайте текст шире 80 знаков - его не удобно будет редактировать и печатать;
* для отступов пользуйтесь табуляцией (клавиша TAB);
* блоки комментариев задавайте с одинаковым отступом. Оптимальной считается такая строка:
<TAB><TAB>mov<TAB>ax,<пробел>bx<(1-3)TAB>;<пробел>текст комментария

Количество табуляций перед комментарием определяется длиной аргументов команды и может быть от 1 до 3. По мере знакомства с синтаксисом языка будут приводиться дополнительные правила.

2. Задание для выполнения

2.1. Запустить эмулятор 8086.
2.2. Пользуясь правилами оформления ассемблерных программ, исправьте слова «Please Register.» на любые понравившиеся (Не забудьте заключить их в апострофы).
2.3. Запустите приложение, нажав кнопку ‘Emulate’ или клавишу F5.
2.4. Запустите полученный код на выполнение, используя кнопку “RUN” или нажмите функциональную клавишу F9.
2.5. Откомпилируйте программу. Вернитесь в главное окно формы, предварительно закрыв все открытые окна, далее нажмите кнопку “Compile”.
2.6. Полученный com-файл запустите во встроенной командной строке WINDOWS 98 на выполнение или запустите сеанс dos в total commander’e.
2.7 Поэкспериментируйте с другими примерами которые открываются при нажатие клавиши “Samples” в главном окне эмулятора.
2.8 Ознакомитесь со встроенной в эмулятор EMU8086 справкой. В ней содержится вся необходимая информация для работы с программой, азы написания программ на языке assembler и др.

3. Контрольные вопросы

3.1. Каковы основные отличия ассемблерных программ от ЯВУ?
3.2. Какова структура ассемблерной программы?
3.3. В чем отличие инструкции от директивы?
3.4. Каковы правила оформления программ на языке ассемблера?
3.5. Каковы этапы получения выполняемого файла?
3.6. Для чего нужен этап отладки программы?
3.7. Опишите основные моменты создания исполняемого файла и эмуляции работы программы?

3.8. Каковы шаги технического создания ассемблерной программы в программах TASM и MASM?

РАЗРАБОТКА ПЕРВОЙ ПРОГРАММЫ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: Знакомство со структурой ассемблерной программы, создание первой программы на языке ассемблера.

1. Структура ассемблерной программы

Чтобы программа выполнилась любой ОС, она должна быть скомпилирована в исполнимый файл. Основные два формата исполнимых файлов в DOS — COM и EXE.

Файлы типа COM содержат только скомпилированный код без какой-либо дополнительной информации о программе Весь код, данные и стек такой полагаются в одном сегменте и не могут превышать 64 Кб.

```
1.  .model tiny
2.  .code
3.  org 100h
4.  begin:
5.      mov ah, 9
6.      mov dx,offset message
7.      int 21h
8.      ret
9.  message db  "Привет", 0dh, 0ah, '$'
10. end begin
```

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

Первая строка определяет модель памяти **TINY**, в которой сегменты кода, данных и стека объединены. Эта модель предназначена для создания файлов типа COM.

В DOS для формирования адреса используется сегмент и смещение.. Для формирования адреса строки **"ПРИВЕТ"** используется пара регистров DS (сегмент) и DX (смещение). При загрузке *.com-программы в память, все сегментные регистры принимают значение равное тому сегменту, в который загрузилась наша программа (в т.ч. и DS). Поэтому нет необходимости загружать в DS сегмент строки (он уже загружен).

Директива **.CODE** начинает сегмент кода, который в нашем случае также должен содержать и данные.

ORG 100h устанавливает значение программного счетчика (IP) в 100h, потому что при загрузке COM-файла в память DOS занимает первые 256 байт (100h) блоком данных PSP и располагает код программы только после этого блока. Все программы, которые компилируются в файлы типа COM, должны начинаться с этой директивы.

Метка **BEGIN:** располагается перед первой командой в программе и будет использоваться в директиве **END** (Begin – англ. начало; end – конец), чтобы указать, с какой команды начинается программа.

Вообще вместо слова **BEGIN** можно было бы использовать что-нибудь другое. Например, **START:**. В таком случае, нам пришлось бы и завершать программу **END START**.

Строки (5) – (7) выводят на экран сообщение “**ПРИВЕТ”**.

Рассмотрим вкратце о регистрах процессора.

Регистр процессора – это специально отведенная память для хранения какого-нибудь числа.

Например:

Если мы хотим сложить два числа, то в математике запишем так:

A=5
B=8
C=A+B.

A, B и C – это своего рода регистры (если говорить о компьютере), в которых могут хранится некоторые данные. A=5 можно прочитать как: Присваиваем A число 5.

Для присвоения регистру какого-нибудь значения, в Ассемблере существует оператор mov (от англ. move – загрузить). Команда **MOV AH,9** помещает число 9 в регистр AH - номер функции DOS «вывод строки».

Команда **MOV DX, OFFSET MESSAGE** помещает в регистр DX смещение метки **MESSAGE** относительно начала сегмента данных, который в нашем случае совпадает с сегментом кода.

OFFSET (по-английски - это смещение). Когда, при ассемблировании, Ассемблер дойдет до этой строки, он заменит **OFFSET MESSAGE** на АДРЕС (смещение) этой строки в памяти. Если мы запишем **OFFSET MESSAGE** (хотя, правильнее будет **MOV DX, WORD OFFSET MESSAGE**), то в **DX** загрузится не адрес (смещение), а первые два символа нашей строки (в данном случае "**Пр**"). Так как **DX** - шестнадцатиразрядный регистр, в него можно загрузить только два байта (один символ всегда один байт).

Команда **INT 21H** вызывает системную функцию DOS (int от англ. interrupt – прерывание). Можно заменить строку **INT 21H** на **INT 33**, программа будет работать корректно. Однако в Ассемблере принято указывать номер прерывания в шестнадцатеричной системе.

Прерывание MS-DOS – это своего рода подпрограмма (часть MS-DOS), которая находится постоянно в памяти и может вызываться в любое время из любой программы.

Эта команда – основное средство взаимодействия программ с операционной системой. В примере вызывается функция DOS номер 9 - вывести строку на экран. Эта функция выводит строку от начала, адрес которого задается в регистрах DS:DX, до первого встречного символа $. При запуске COM-файла регистр DS автоматически загружается сегментным адресом программы, а регистр DX был подготовлен предыдущей командой.

Рассмотрим вышесказанное на примере (мелким шрифтом выделим примечания):

Программа сложения двух чисел
НачалоПрограммы
A=5 в переменную A заносим значение 5
B=8 в переменную B значение 8
ВызовПодпрограммы Сложение
теперь C равно 13
A=10 тоже самое, только другие числа
B=25
ВызовПодпрограммы Сложение
теперь C равно 35
КонецПрограммы
…
Подпрограмма Сложение
C=A+B
ВозвратИзПодпрограммы возвращаемся в то место, откуда вызывали
КонецПодпрограммы

В данном примере мы дважды вызвали подпрограмму Сложение, которая сложила два числа, переданные ей в переменных A и B. Результат помещается в переменную C. Когда вызывается подпрограмма, компьютер запоминает с какого места она была вызвана, а затем, когда закончила работу подпрограмма, компьютер возвращается в то место, откуда она вызывалась. Т.о. можно вызывать подпрограммы неопределенное количество раз с любого места.

Команда **RET** пользуется обычно для возвращения из процедуры. DOS вызывается COM-программы так, что команда **RET** корректно завершает программу.

DOS при вызове COM-файла помещает в стек сегментный адрес программы и ноль, так что **RET** передает управление на нулевой адрес текущего сегмента, то есть на первый байт PSP. Там находится код команды **INT 20H,** которая и используется для возвращения управления в DOS. Можно сразу заканчивать программу командой INT 20h, хотя это длиннее на 1 байт.

Следующая строка примера определяет строку данных, содержащую текст **“ПРИВЕТ”** управляющий символ ASCII **возврат каретки** с кодом **ODh**, управляющий символ ASCII **перевод строки** с кодом **0Ah** и символ **$** завершающий строку (если мы его уберем, то 21h прерывание продолжит вывод до тех пор, пока не встретится где-нибудь в памяти символ $, на экране мы увидим “мусор). Первое слово (message – сообщение) – название сообщения. Оно может быть любым (например, mess или string и пр.).

Управляющие символы (**ODh** и **0Ah**) переводят курсор на первую позицию следующей строки.

Директива **END** завершает программу, одновременно указывая, с какой метки должно начинаться ее выполнение.

В качестве дополнительного примера создадим еще одну строку, которую назовем **message1**. Затем, начиная со строки (9) вставим следующие команды

```
 9        mov dx,offset message1
10                int 21h
11                int 20h
12   message      db   "Привет", 0dh, 0ah, '$'
13   message1     db   "Группа", 0dh, 0ah, '$'
14   end begin
```

и скомпилируем программу заново.

2. Задание для выполнения

2.1. Запустить эмулятор EMU8086.
2.2. Пользуясь правилами оформления ассемблерных программ, наберите код, приведенный в примере 1, запустите код на выполнение.
2.3. Откомпилируйте пример №2;
2.4. Вернитесь в главное окно формы, предварительно закрыв все открытые окна, далее нажмите кнопку “Compile”.
2.5. Полученный com-файл запустите в сеансе dos.
2.6. Создайте на языка Pascal программу выводящую на экран слово **“Привет”** и сравните размеры получаемых файлов (Pascal и Assembler).

3. Контрольные вопросы

3.1. Характеристика структуры файла типа *.com?
3.2. Какова структура ассемблерной программы?
3.3. С какой целью в код программы на ассемблере для DOS вводится строка **ORG 100h**?
3.4. Назначение команды **MOV** ?
3.5. Прерывания **21h** и **20h.** Назначение?
3.6 Сущность и целесообразность использования команды **RET** вместо прерывания **20h** ?
3.6. Символ **‘$’** методика применения?

3.7. Связка “**BEGIN: – END BEGIN**”. Правила применения?

ЛАБОРАТОРНАЯ РАБОТА №3 - 4

СТРУКТУРА ИСПОЛНИМЫХ ФАЙЛОВ ТИПА *.EXE. ПРОСТЫЕ АРИФМЕТИЧЕСКИЕ ДЕЙСТВИЯ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: *Изучение принципов составления простейших*.exe программ. Изучение приемов работы с простейшими операторами арифметических действий.*

1. Краткие теоретические сведения (программа типа *.EXE).

Файлы типа EXE содержат заголовок, в котором описывается размер файла, требуемый объем памяти, список команд в программе, использующих абсолютные адреса, которые зависят от расположения программы в памяти, и т.д. EXE-файл может иметь любой размер. Формат EXE также используется для исполнимых файлов в различных версиях DOS-расширителей и Windows, но со значительными изменениями.

Операционная система DOS не использует расширения для определения типа файла. Первые два байта заголовка EXE-файла — символы «MZ» или «ZM», и если файл начинается с этих символов и длиннее некоторого порогового значения, разного для разных версий DOS, он загружается как EXE, если нет — как COM.

EXE-программы немного сложнее в исполнении, но для них отсутствует ограничение размера в 64 килобайта, так что все достаточно большие программы используют именно этот формат. Конечно, ассемблер позволяет уместить и в 64 килобайтах весьма сложные и большие алгоритмы, а все данные хранить в отдельных файлах, но ограничение размера все равно очень серьезно, и даже чисто ассемблерные программы могут с ним сталкиваться.

Простой пример EXE-файла :

```
11.   .model small                    ; сегмент стека размером в 256 байт
12.   .stack 100h                     ; сегмент стека размером в 256 байт
13.   .code                           ; сегмент кода, который содержит и
                                      данные.
14.   Begin:                          ; метка начала кода программы
15.       mov ax,@data      ;         ; сегментный адрес строки message
16.       mov  ds,ax                  помещается в DS
17.       mov dx,offset string        помещает в регистр DX смещение метки
                                      String относительно начала сегмента
                                      данных
18.       mov ah,9                    ; помещаем номер функции DOS «вывод
                                      строки (9)» в регистр AH.
19.       int 21h                     ; функция DOS "вывод строки"
20.       mov ax,4C00h                ; завершение программы типа - exe
21.       int 21h                     ; функция DOS "завершить программу"
22.   .data                           ; начало сегмента данных
23.     string db "Privet", 0Dh,0Ah,'$'  ; строка с содержащая выводимые
                                      данные.
24.   end begin                       ; метка окончания  кода программы
```

В примере определяются три сегмента — сегмент стека директивой **.STACK** размером в 256 байт, сегмент кода, начинающийся с директивы **.CODE**, и сегмент данных,

начинающийся с **.DATA**. При запуске EXE-программы регистр DS уже не содержит адреса сегмента со строкой string (он указывает на сегмент, содержащий блок данных PSP), а для вызова используемой функции DOS этот регистр должен иметь сегментный адрес строки. Команда **MOV AX,@DATA** загружает в **AX** сегментный адрес группы сегментов данных **@DATA**, а **MOV DS,AX** копирует его в **DS**. Программы типа EXE должны завершаться системным вызовом **DOS 4Ch:** в регистр **AH** помещается значение **4Ch**, в регистр **AL** помещается код возврата (в данном примере код возврата 0 и регистры **AH** и **AL** загружаются одной командой **MOV AX,4C00h**), после чего вызывается прерывание **21h**.

2. Простые арифметические операторы.

Арифметические команды любого микропроцессора привлекают к себе наибольшее внимание. Каждый заинтересован в выполнении арифметических вычислений, и именно эти команды проделывают такую работу. Хотя их немного, они выполняют большинство преобразований данных, а микропроцессоре. В реальных же условиях арифметические команды занимают лишь малую часть всех исполняемых команд.

2.1. Сложение.

Команда **ADD** (Addition – сложение (гл. to add – сложить)) осуществляет сложение первого и второго операндов. Исходное значение первого операнда (приемника) теряется, замещаясь результатом сложения. Второй операнд не изменяется. В качестве первого операнда команды **ADD** можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команду **ADD** можно использовать для сложения как обычных целых чисел, так и двоично-десятичных (с использованием регистра **AX** для хранения результата). Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

Команда	Назначение	Процессор
ADD приемник, источник	Сложение	8086

Примеры:

```
mov al,10 ---> загружаем в регистр AL число 10
add al,15 ---> al = 25; al - приемник, 15 - источник

mov ax,25000 ---> загружаем в регистр AX число 25000
add ax,10000 ---> ax = 35000; ax - приемник, 10000 - источник

mov cx,200 ---> загружаем в регистр CX число 200
mov bx,760 ---> a в регистр BX --- 760
add cx,bx ---> cx = 960, bx = 760 (bx не меняется); cx - приемник, bx - источник
```

2.2. Вычитание.

Команда **SUB** (**Subtraction** – вычитание) вычитает второй операнд (источник) из первого (приемника) и помещает результат на место первого операнда. Исходное значение первого операнда (уменьшаемое) теряется. Таким образом, если команду вычитания записать в общем виде

SUB операнд1, операнд2

то ее действие можно условно изобразить следующим образом:

операнд1 - операнд2 -> операнд1

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда воздействует на флаги **OF, SF, ZF, AF, PF** и **CF**.

Команда	Назначение	Процессор
SUB приемник, источник	Вычитание	8086

Примеры:
mov al,10
sub al,7 ---> al = 3; al - приемник, 7 - источник

mov ax,25000
sub ax,10000 ---> ax = 15000; ax - приемник, 10000 - источник

mov cx,100
mov bx,15
sub cx,bx ---> cx = 85, bx = 15 (bx не меняется); cx - приемник, bx - источник

2.3. Инкремент (увеличение на 1).

Команда INC (Increment – инкремент) прибавляет 1 к операнду, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги OF, SF, ZF, AF и PF. Команда не воздействует на флаг CF; если требуется воздействие на этот флаг, необходимо использовать команду Add Op,l.

Команда INC (Increment – инкремент) увеличивает на единицу регистр или значение операнда в памяти.

Она эквивалентна команде ADD источник, 1 только выполняется гораздо быстрее.

Команда	Назначение	Процессор
INC приемник	Увеличение на единицу	8086

Примеры:
mov al,15
inc al ---> теперь AL = 16 (эквивалентна add al,1)

mov dh,39h
inc dh ---> DH = 3Ah (эквивалентна add dh,1)

mov cl,4Fh
inc cl ---> CL = 50h (эквивалентна add cl,1)

2.4. Декремент (уменьшение на 1).

Команда **DEC** (Decrement – декремент) вычитает 1 из операнда, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги **OF, SF, ZF**, **AF** и **PF**.

Она эквивалентна команде **SUB** источник, **1** только выполняется гораздо быстрее.

Команда	Назначение	Процессор
INC приемник	Увеличение на единицу	8086

Примеры:

mov al,15
dec al ---> теперь AL = 14 (эквивалентна sub al,1)

mov dh,39h
dec dh ---> DH = 38h (эквивалентна sub dh,1)

mov cl,4Fh
dec cl ---> CL = 4Dh (эквивалентна sub cl,1)

3. Задание для выполнения.

3.1. Запустить эмулятор EMU8086.
3.2. Получите задание у преподавателя (один из пяти вариантов табл.№1) и, пользуясь правилами оформления ассемблерных программ, напишите программы расчета значения **A** (два-три варианта).
2.3. Программу ассемблируйте в файл типа *.exe;

3. Контрольные вопросы
3.1. Структура файлов типа *.exe?
3.2. Структурные отличия файлов *.exe от *.com в операционной среде DOS?
3.3. Команда **add** основное назначение?
3.4. Команда **sub** основное назначение?
3.5. Команда **inc** основное назначение?
3.6. Команда **dec** основное назначение?

Табл. №1

№ вар.	Расчетная формула	B	C	D
1.	**A=B+C-D**	**1**	**35**	**23**
2.	**A=B+C+D**	**65**	**1**	**1**
3.	**A=C-D+B**	**1**	**33**	**1**
4.	**A=D+A-B**	**18**	**1**	**88**
5.	**A= B-C+D**	**45**	**10**	**1**

По согласованию с преподавателем можно изменить как расчетную формулу, так и значения коэффициентов (**B, C, D**).

СПОСОБЫ АДРЕСАЦИИ НА ЯЗЫКЕ АССЕМБЛЕРА

Цель работы: *Изучить основные способы адресации*.

3. Способы адресации.

Способом, или режимом адресации называют процедуру нахождения операнда для выполняемой команды. Если команда использует два операнда, то для каждого из них должен быть задан способ адресации, причем режимы адресации первого и второго операнда могут, как совпадать, так и различаться. Операнды команды могут находиться в разных местах: непосредственно в составе кода команды, в каком-либо регистре, в ячейке памяти; в последнем случае существует несколько возможностей указания его адреса. Строго говоря, способы адресации являются элементом архитектуры процессора, отражая заложенные в нем возможности поиска операндов. С другой стороны, различные способы адресации определенным образом обозначаются в языке ассемблера и являются разделом языка.

В программах, написанных на языке ассемблера термин "операнд" применим к обозначению тех физических объектов, с которыми имеет дело процессор при выполнении машинной команды и, говоря об операндах команд языка, понимают в действительности операнды машинных команд. По отношению к командам ассемблера используется термин "параметры".

В архитектуре современных 32-разрядных процессоров Intel предусмотрены довольно изощренные способы адресации; в МП 86 способов адресации меньше. Мы в настоящей лабораторной работе ознакомимся с режимами адресации, используемые в МП 86.

Различают следующие режимы адресации:

- регистровый;
- непосредственный;
- прямой;
- регистровый косвенный (базовый или индексный);
- регистровый косвенный со смещением (базовый или индексный);
- базово-индексный;
- базовый индексный со смещением.

1.1 Регистровый режим

Значение операнда-источника предварительно запоминается в одном из встроенных регистров микропроцессора. Сам регистр становится эффективным адресом. Операнд (байт или слово) находится в регистре. Этот способ применим ко всем программно-адрессуемым регистрам процессора:

25.	**inc CX**	; Увеличение на 1 содержимого **CX**
26.	**push DS**	; Сегментный адрес сохраняется в стеке
27.	**xchg BX,BP**	; Регистры **BX** и **BP** обмениваются содержимым
28.	**mov ES,AX**	; Содержимое **AX** пересылается в **ES**

1.2 Непосредственный режим

Непосредственная адресация. Операнд (байт или слово) указывается в команде и после трансляции поступает в код команды; он может иметь любой смысл (число, адрес, код ASCII), а также быть представлен в виде символического обозначения.

1.	**mov AH, 40h**	; Число 40h загружается в AH

2.	**mov AL,'*'**	;Код ASCII символа "*' загружается в AL
3.	**int 21h**	;Команда прерывания с аргументом 21h
4.	**limit equ 528**	;Число 528 получает обозначение limit
5.	**mov CX,limit**	;Число, обозначенное limit, загружается в CX

Команда **mov**, использованная в последнем предложении, имеет два операнда; первый операнд определяется с помощью регистровой адресации, второй - с помощью непосредственной.

Важным применением непосредственной адресации является пересылка относительных адресов (смещений), для этого используется описатель offset (смещение):

;Сегмент данных
string db "Privet" ;Строка символов

;Сегмент команд
mov DX,offset string ;Адрес строки засылается в DX

1.3 Прямой режим.

Адресуется память; адрес ячейки памяти (слова или байта) указывается в команде (обычно в символической форме) и поступает в код команды:

;Сегмент данных
meml dw 0 ;Слово памяти содержит 0
mem2 db 230 ;Байт памяти содержит 230

;Сегмент команд
inc meml ;Содержимое слова meml увеличивается на 1
mov DX, meml ;Содержимое слова с именем menu загружается в DX
mov AL,mem2 ;Содержимое байта с именем mem2 загружается в AL

Сравнивая этот пример с предыдущим, мы видим, что указание в команде имени ячейки памяти обозначает, что операндом является содержимое этой ячейки; указание имени ячейки с описателем **offset** - что операндом является адрес ячейки.

Прямая адресация памяти на первой взгляд, кажется, простой и наглядной. Если мы хотим обратиться, например, к ячейке **meml**, мы просто указываем ее имя в программе. В действительности, однако, дело обстоит сложнее. Адрес любой ячейки состоит из двух компонентов: сегментного адреса и смещения. Обозначения **meml** и **mem2** в предыдущем примере, являются смещениями. Сегментные же адреса хранятся в сегментных регистрах. Однако сегментных регистров четыре: **DS**, **ES, CS** и **SS**. Каким образом процессор узнает, из какого регистра взять сегментный адрес, и как сообщить ему об этом в программе?

Процессор различает группу кодов, носящих название префиксов. Имеется несколько групп префиксов: повторения, размера адреса, размера операнда, замены сегмента. Здесь нас будут интересовать префиксы замены сегмента.

Команды процессора, обращающиеся к памяти, могут в качестве первого байта своего кода содержать префикс замены сегмента, с помощью которого процессор определяет, из какого сегментного регистра взять сегментный адрес. Для сегментного регистра **ES** код префикса составляет **26h**, для **SS - 361i**, для **CS - 2Eh**. Если префикс отсутствует, сегментный адрес берется из регистра **DS** (хотя для него тоже предусмотрен свой префикс).

В приведенном примере, по умолчанию, все данные адресуются через сегментный регистр **DS**, так что вместо i**nc meml** можно было написать **inc DS:mem.** В случае замены сегментного регистра его обязательно нужно указывать явно:

inc ES:mem1
inc CS:mem2

Обращение к ячейке памяти по известному абсолютному адресу осуществляется следующим образом:

mov AL,DS:[17h]	Загрузка в AL содержимого ячейки со смещением 17h в сегменте, определяемом содержимым DS

1.4 Регистровый косвенный (базовый и индексный).

Адресуется память (байт или слово). Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в прямые скобки. В МП 86 косвенная адресация допустима только через регистры **BX, BP, SI** и **DI**. При использовании регистров **BX** или **BP** адресацию называют базовой, при использовании регистров **SI** или **DI** - индексной.

Если косвенная адресация осуществляется через один из регистров **BX, SI** или **DI**, то подразумевается сегмент, адресуемый через **DS**, поэтому при адресации через этот регистр обозначение **DS:** можно опустить:

mov es:[bx],'1' ————→ mov [bx],'1'

Кстати, этот фрагмент немного эффективнее предыдущего в смысле расходования памяти. Из-за отсутствия в коде последней команды префикса замены сегмента он занимает на 1 байт меньше места.

Регистры **BX, SI** и **DI** в данном применении совершенно равнозначны, и с одинаковым успехом можно воспользоваться любым из них:

Не так обстоит дело с регистром **BP**. Этот регистр специально предназначен для работы со стеком, и при адресации через этот регистр в режимах косвенной адресации подразумевается сегмент стека; другими словами, в качестве сегментного регистра по умолчанию используется регистр **SS**.

Обычно косвенная адресация к стеку используется в тех случаях, когда необходимо обратиться к данным, содержащимся в стеке, без изъятия их оттуда (например, если к эти данные приходится считывать неоднократно).

Обозначение этого способа адресации:

[BX]	(подразумевается **DS:[BX]**)
[BP]	(подразумевается **SS:[BP]**)
[SI]	(подразумевается **DS:[SI]**)
[DI]	(подразумевается **DS:[DI]**)

Использование базовой адресации, на первый взгляд, снижает эффективность программы, так как требует дополнительной операции - загрузки в базовый регистр требуемого адреса. Однако команда с базовой адресацией занимает меньше места в памяти (так как в нее не входит адрес ячейки) и выполняется быстрее команды с прямой адресацией (из-за того, что команда короче, процессору требуется меньше времени на ее считывание из памяти). Поэтому базовая адресация эффективна в тех случаях, когда по заданному адресу приходится обращаться многократно, особенно, в цикле. Выигрыш оказывается тем больше, чем большее число, раз происходит обращение по указанному адресу. С другой стороны,

возможности этого режима адресации невелики, и на практике чаще используют более сложные способы.

Примеры:

1.	**mov SI, offset string**	; В **SI** загружается относительный адрес ячейки string
2.	**mov AX,[SI]**	; Содержимое ячейки string загружается в **AX**
3.	**inc [SI]**	; Увеличиваться содержимое ячейки string
4.	**mov BX,[SI]**	; Новое содержимое ячейки string загружается в **BX**
5.	**mov DI, SI**	; Относительный адрес ячейки string копируется в **DI**

1.5 Регистровый косвенный режим со смещением (базовый и индексный).

Адресуется память (байт или слово). Относительный адрес операнда определяется, как сумма содержимого регистра **BX, BP, SI** или **DI** и указанной в команде константы, иногда называемой смещением. Смещение может быть числом или адресом. Так же, как и в случае базовой адресации, при использовании регистров **BX, SI** и **DI** подразумевается сегмент, адресуемый через **DS**, а при использовании **BP** подразумевается сегмент стека и, соответственно, регистр **SS.**

смещение = {SP, BP, DI, SI, BX} + смещение из команды

Иногда можно встретиться с альтернативными обозначениями того же способа адресации, которые допускает ассемблер. Вместо, например, 4[BX] можно с таким же успехом написать [BX+4], 4+[BX] или [BX]+4. Такая неоднозначность языка ничего, кроме путаницы, не приносит, однако ее надо иметь в виду, так как с этими обозначениями можно столкнуться, например, рассматривая текст деассемблированной программы.

Рассмотрим теперь пример использования базовой адресации со смещением при обращении к стеку:

смещение = {SP, BP, DI, SI, BX} + смещение из команды

Здесь квадратные скобки **[ ]** - это тоже оператор. Он вычисляет адрес как сумму того, что находится внутри скобок с тем, что находится снаружи.

array db 0, 10, 20, 30, 40, 50, 60 ;Пусть в сегменте данных определен массив:

Последовательность команд:
mov BX,5
mov AL,array[5] **;**загрузит в AL элемент массива с индексом 5, то есть 50.

Тот же результат будет получен и в таких последовательностях команд:
mov BX,offset array
mov AL,5[BX]

или
mov AL,[BX]+5
mov AL,[BX+5]

1.6 Базово-индексный режим

Адресуется память (байт или слово). Относительный адрес операнда определяется, как сумма содержимого следующих пар регистров:

смещение **[BX][SI]**	(подразумевается DS:*смещение* [BX][SI])
смещение **[BX][DI]**	(подразумевается DS:*смещение* [BX][DI])

***смещение* [BP][SI]**	(подразумевается SS:*смещение* [BP][SI])
***смещение* [BP][DI]**	(подразумевается SS:*смещение* [BP][DI])

Во всех этих случаях можно также писать:

***смещение* [BX+SI]**
[*смещение* +BX+SI]
[BX+SI]+*смещение*

Это чрезвычайно распространенный способ адресации, особенно, при работе с массивами. В нем используются два регистра, при этом одним из них должен быть базовый (**BX** или **BP**), а другим - индексный (**SI** или **DI**). Как правило, в одном из регистров находится адрес массива, а в другом - индекс в нем, при этом совершенно безразлично, в каком что.

1.7 Базово-индексная адресация со смещением.

Адресуется память (байт или слово). Относительный адрес операнда определяется как сумма содержимого двух регистров и смещения.

Это способ адресации является развитием предыдущего. В нем используются те же пары регистров, но полученный с их помощью результирующий адрес можно еще сместить на значение указанной в команде константы. Как и в случае базово-индексной адресации, константа может представлять собой индекс (и тогда в одном из регистров должен содержаться базовый адрес памяти), но может быть и базовым адресом. В последнем случае регистры могут использоваться для хранения составляющих индекса.

Приведем формальный пример рассматриваемого режима адресации.

Пусть в сегменте данных определен массив из 24 байт

```
syms db 'ЙЦУКЕНГШЩЗХЪ'
     db 'йцукенгшщзхъ'
```

Последовательность команд

```
mov BX,12mov SI,6
mov DL,syms[BX][SI] ;загрузит в регистр DL элемент с индексом 6 из второго ряда,
                     то есть код ASCII буквы г
```

Тот же результат будет получен и в таком варианте:

```
mov BX,offset syms
mov SI,6
mov DL,12[BX][SI]
```

2. Порядок выполнения работы:

1. С помощью редактора эмулятора EMU 8086 напишите программу, исходный текст которой приводится в листинге №1:
2. Создайте исполняемый файл типа **MZ**.
3. Изучите структуру программы, также изучите структуру сегмента данных программы: найдите в нем все переменные, определенные в тексте программы.

4. Переделайте программу с использованием упрощенных директив сегментации так, чтобы получить исполняемый файл типа .COM и сравните размеры программ.

5. Выполните первые 5 шагов программы, анализируя и записывая состояние регистров на каждом шаге.

6. Занесите в CX 00FFh. Определите по способу адресации ячейку памяти в сегменте, где произойдут изменения, записать ее адрес.

7. Выполните дальнейшие шаги программы, анализируя возможные способы адресации.

8. Подготовьте отчет, который должен содержать тексты программ, адреса сегментных регистров и записи адресов ячеек памяти против соответствующих команд, а также запись содержимого этих ячеек.

9. В отчете должны содержаться ответы на следующие вопросы.

3. Контрольные вопросы

1. Как переслать содержимое X в Y?
2. Чем отличаются команды

   MOV [si], cx

   и

   MOV si, cx?
3. К какому способу адресации относится команда MOV dx, offset message?
4. Какие сегменты используются при следующих вариантах адресации: [BX][SI], [BX][DI], [BP][SI], [BP][DI] ?

5. Что произойдет при выполнении инструкции

   MOV AL, DS: 17h?

Чем эта команда отличается от следующей:

   MOV AL, DS: [17h] ?

6. Пусть в сегменте данных определен массив

   Array db 0,15,22,31,44,45,62,67,76,99

Что окажется в регистре AL после выполнения команд:

   MOV BX, 5

   MOV AL, array[BX]?

7. Какой это способ адресации (пример вопроса 6)?

8. Укажите, какие инструкции в программе (листинг №1), созданной в данной лабораторной работе, относятся к инструкциям:

- с непосредственным;
- косвенным режимом адресации?

9. Укажите способ записи обращения напрямую к ячейке памяти по известному абсолютному адресу?

10. Префиксы, Виды префиксов. Префиксы замены сегмента?

11.Перечислите регистры косвенной и базовой адресации. Опишите отличия?

12. Сущность эффективности базовой адресации в сравнении с прямой?

Листинг №1.

```
TITLE MOVE2
MOVE2 SEGMENT 'CODE'
ASSUME CS:MOVE2, DS:DATA
MYPROC PROC
OUTPROC:
MOV AX,DATA
MOV DS,AX
MOV AH,BH
MOV AH,X
MOV CH,3
MOV AX,3
MOV AX,Y
MOV [SI],CX
MOV [BP],CX
MOV [SI],258
MOV [BP+516],1027
MOV BYTE PTR X,255
MOV BYTE PTR [DI+515],4
MOV WORD PTR [DI+515],4
MOV [DI+BP+515],258
MOV AX,[SI+BX+258]
MOV AH,4CH
INT 21H
MYPROC ENDP
MOVE2 ENDS
DATA SEGMENT
X DB 1
Y DW 2
DATA ENDS
END MYPROC
```

ЛАБОРАТОРНАЯ РАБОТА № 5 - 6

ВЫВОД НА ЭКРАН В ТЕКСТОВОМ РЕЖИМЕ

Цель работы: *Ознакомится с основными средствами вывода текстовых данных на экран посредством средств операционной системы DOS, средствами BIOS и средствами непосредственного (прямого) отображением в видеобуфер.*

4. Средства DOS.

1.1 Функция DOS 02h.

Функция **DOS 02h** — Записать символ в **STDOUT** с проверкой на Ctrl-Break

Ввод:	**AH** = **02h** **DL** = ASCII-код символа
Вывод:	Никакого, согласно документации, но на самом деле: **AL** = код последнего записанного символа (равен DL, кроме случая, когда DL = 09h (табуляция), тогда в AL возвращается 20h).

Эта функция при выводе на экран обрабатывает некоторые управляющие символы — вывод символа **BEL (07h)** приводит к звуковому сигналу, символ **BS (08h)** приводит к движению курсора влево на одну позицию, символ **HT (09h)** заменяется на несколько пробелов, символ **LF (0Ah)** опускает курсор на одну позицию вниз, и **CR (0Dh)** приводит к переходу на начало текущей строки.

Если в ходе работы этой функции была нажата комбинация клавиш Ctrl-Break, вызывается прерывание **23h**, которое по умолчанию осуществляет выход из программы.

*Простой пример работы функции **DOS 02h.***

Пример № 1.1

29.	**.model tiny**	; модель памяти в которой сегменты кода, данных и стека объединены.
30.	**.code**	; сегмент кода, который содержит данные.
31.	**org 100h**	; начало COM-файла
32.	**begin:**	; метка начала кода программы
33.	**mov dl,< ASCII-код символа >**	; заносим в регистр **dl** – любой ASCII-код символа
34.	**mov ah,2**	; номер функции DOS "вывод символа"
35.	**int 21h**	; вызов DOS
36.	**ret**	; функция DOS "завершить программу"
37.	**end begin**	; метка окончания кода программы

Эта программа, выводит на экран любой ASCII-символ, в установленную позицию курсора.

Все функции DOS вывода на экран используют устройство **STDOUT**, стандартный вывод. Это позволяет перенаправлять вывод программы в файл или на стандартный ввод другой программы. Например, если откомпилировать приведен пример (создать файл **cod.com**) и написать в командной строке

cod.com > cod.out

то на экран ничего выдано не будет, а в текущем каталоге появится файл **cod.out**, содержащий ASCII-код символа.

1. Функция DOS 06h.

Функция **DOS 06h** — Записать символ в **STDOUT** без проверки на Ctrl-Break

Ввод:	**AH** = 06h **DL** = ASCII-код символа (кроме FFh)
Вывод:	Никакого, согласно документации, но на самом деле: AL = код записанного символа (копия DL)

Эта функция не обрабатывает управляющие символы (**CR, LF, HT** и **BS** выполняют свои функции при выводе на экран, но сохраняются при перенаправлении вывода в файл) и не проверяет нажатие Ctrl-Break.

Заменим в примере № 1.1 **MOV AH,2** на **MOV AH,6** и перекомпилируем этот пример. Работу откомпилированного примера смотрим в операционной системе **MS-DOS.**

2. Функция DOS 09h

Функция **DOS 09h** — Записать строку в **STDOUT** с проверкой на Ctrl-Break

Ввод:	**AH** = 09h **DS:DX** = адрес строки, заканчивающейся символом $ (24h)
Вывод:	Никакого, согласно документации, но на самом деле: AL = 24h (код последнего символа)

Действие этой функции полностью аналогично действию функции **02h**, но выводится не один символ, а целая строка (смотри лабораторную работу №2).

1.4 Функция DOS 40h

Функция **DOS 40h** — Записать в файл или устройство

Ввод:	**AH** = 40h **BX** = 1 для **STDOUT** или 2 для **STDERR** **DS:DX** = адрес начала строки **CX** = длина строки
Вывод:	**CF** = 0, **AX** = число записанных байт

Эта функция предназначена для записи в файл, но, если в регистр **BX** поместить число 1, функция **40h** будет выводить данные на **STDOUT**, а если **BX** = 2 — на устройство **STDERR**. **STDERR** всегда выводит данные на экран и не перенаправляется в файлы. На этой функции основаны используемые в C функции стандартного вывода — фактически функция C **fputs**() просто вызывает это прерывание, помещая свой первый аргумент в **BX**, адрес строки (второй аргумент) — в **DS:DX** и длину — в **CX**.

*Простой пример работы функции **DOS 40h.***

Пример № 1.2

1	**.model tiny**	; модель памяти в которой сегменты кода, данных и стека объединены.
2	**.code**	; сегмент кода, который содержит данные.
3	**org 100h**	; начало COM-файла
4	**begin:**	; метка начала кода программы
5	**mov ah,40h**	; номер функции DOS

6	**mov bx,2**	; указываем устройство **STDERR**
7	**mov dx,offset message**	; **DS:DX** - адрес строки
8	**mov cx,25**	; CX - длина строки
9	**int 21h**	; вызов DOS
10	**ret**	; функция DOS "завершить программу"
11	**message db "This function can print $"**	**;** строка с содержащая выводимые данные.
12	**end begin**	; метка окончания кода программы

Если скомпилировать этот пример и запустить ее командой
dosout.com > dosout.out
то сообщение появится на экране, а файл dosout2.out окажется пустым.

1.5 Прерывание INT 29H

INT 29h: Быстрый вывод символа на экран

Ввод:	**AL** = ASCII-код символа

Простой пример работы прерывания INT 29h.

Пример № 1.3

1.	**.model tiny**	; модель памяти в которой сегменты кода, данных и стека объединены.
2.	**.code**	; сегмент кода, который содержит данные.
3.	**org 100h**	; начало COM-файла
4.	**begin:**	; метка начала кода программы
5.	**mov ax, < ASCII-код символа >**	; заносим в регистр **ax** – любой ASCII-код символа
6.	**int 29h**	; вызов прерывания DOS – вызов символа;
7.	**ret**	; функция DOS "завершить программу"
8.	**end begin**	; метка окончания кода программы

В большинстве случаев **INT 29h** просто немедленно вызывает функцию BIOS «вывод символа на экран в режиме телетайпа», так что никаких преимуществ, кроме экономии байт при написании как можно более коротких программ, она не имеет.

2. Средства BIOS

Функции DOS вывода на экран позволяют перенаправить вывод в файл, но не позволяют вывести текст в любую позицию экрана и не позволяют изменить цвет текста. DOS предполагает, что для более тонкой работы с экраном программы должны использоваться видеофункции BIOS. BIOS — обеспечивает доступ к некоторым устройствам, в частности к видеоадаптеру. Все функции видеосервиса BIOS вызываются через прерывание 10h.

2.1. Выбор видеорежима

BIOS предоставляет возможность переключения экрана в различные текстовые и графические режимы. Режимы отличаются друг от друга разрешением (для графических) и количеством строк и столбцов (для текстовых), а также количеством возможных цветов.

2.1.1. Стандартные видеорежимы

INT 10h, AH = 00 — Установить видеорежим

Ввод:	**AL** = номер режима в младших 7 битах
Вывод:	Обычно никакого, но некоторые BIOS (Phoenix и AMI) помещают в AL 30H для текстовых режимов и 20h для графических

Пример работы.

Пример № 2.1

1.	**.model tiny**	; модель памяти в которой сегменты кода, данных и стека объединены.
2.	**.code**	; сегмент кода, который содержит данные.
3.	**org 100h**	; начало COM-файла
4.	**begin:**	; метка начала кода программы
5.	**mov ah,00**	; устанавливаем видеорежим
6.	**mov al,5**	; устанавливаем номер режима
7.	**int 10h**	; вызов прерывания DOS – вызов видеосервиса;
8.	**ret**	; функция DOS "завершить программу"
9.	**end begin**	; метка окончания кода программы

Вызов этой функции приводит к тому, что экран переводится в выбранный режим. Если старший бит AL не установлен в 1, экран очищается. Номера текстовых режимов — 0, 1, 2, 3 и 7. 0 и 1— 16-цветные режимы 40x25 (с 25 строками по 40 символов в строке), 2 и 3 — 16-цветные режимы 80x25, 7 — монохромный режим 80x25. Существует еще много текстовых режимов с более высоким разрешением (80x43, 80x60, 132x50 и т.д.), но их номера для вызова через эту функцию различны для разных видеоадаптеров (например, режим 61h — 132x50 для Cirrus 5320 и 132x29 для Genoa 6400). Однако, если видеоадаптер поддерживает стандарт VESA BIOS Extention, в режимы с высоким разрешением можно переключаться, используя функцию 4Fh.

2.1.2. SuperVGA-видеорежим

INT 10h, AH = 4Fh, AL = 02 — Установить SuperVGA-видеорежим

Ввод:	**BX** = номер режима в младших 13 битах
Вывод:	**AL** = **4Fh**, если эта функция поддерживается **AH** = 0, если переключение произошло успешно **AH** = 1, если произошла ошибка

Если бит 15 регистра **BX** установлен в 1, видеопамять не очищается. Текстовые режимы, которые можно вызвать с использованием этой функции: 80x60 (режим **108h**), 132x25 (**109h**), 132x43 (**10Ah**), 132x50 (**10Bh**), 132x60 (**10Ch**).

Видеорежим, используемый в DOS по умолчанию, — текстовый режим 3.

2.2 Управление положением курсора

2.2.1. Устанавливаем положение курсора

INT 10h, AH = 02 — Установить положение курсора

Ввод:	**AH** = 02 **BH** = номер страницы **DH** = строка **DL** = столбец

Пример работы.

Пример № 2.2.1

1.	**.model tiny**	; модель памяти, в которой сегменты кода, данных и стека объединены.
2.	**.code**	; сегмент кода, который содержит данные.
3.	**org 100h**	; начало COM-файла
4.	**begin:**	; метка начала кода программы
5.	**mov ah,02**	; устанавливаем положение курсора
6.	**mov bh,0**	; устанавливаем номер страницы
7.	**mov dh,12**	; строка 12
8.	**mov dl,29**	; столбец 29
9.	**int 10h**	; прерывания DOS – установить положение курсора в точку 12,29
10.	**mov ax, < ASCII-код символа >**	; заносим в регистр **ax** – любой ASCII-код символа
11.	**int 29h**	; вызов прерывания DOS – вызов символа;
12.	**ret**	; функция DOS "завершить программу"
13.	**end begin**	; метка окончания кода программы

С помощью этой функции можно установить курсор в любую позицию экрана, и дальнейший вывод текста будет происходить из этой позиции. Отсчет номера строки и столбца ведется от верхнего левого угла экрана (символ в левой верхней позиции имеет координаты 0,0). Номера страниц 0 – 3 (для режимов 2 и 3)и 0 – 7 (для режимов 1 и 2) соответствуют области памяти, содержимое которой в данный момент отображается на экране. Можно вывести текст в неактивную в настоящий момент страницу, а затем переключиться на нее, чтобы изображение изменилось мгновенно.

2.2.2 Считываем положение и размер курсора

INT 10h, AH = 03 — Считать положение и размер курсора

Ввод:	**AH** = 03 **BH** = номер страницы
Вывод:	**DH, DL** = строка и столбец текущей позиции курсора **CH, CL** = первая и последняя строки курсора

Возвращает текущее состояние курсора на выбранной странице (каждая страница использует собственный независимый курсор).

2.3. Вывод символов на экран

Каждый символ на экране описывается двумя байтами — ASCII-кодом символа и байтом атрибута, указывающим цвет символа и фона, а также является ли символ мигающим.

Атрибут символа:

Бит 7: символ мигает (по умолчанию) или фон яркого цвета (если его действие было переопределено видеофункцией 10h).

Биты 6 – 4: цвет фона.

Бит 3: символ яркого цвета (по умолчанию) или фон мигает (если его действие было переопределено видеофункцией 11h).

Биты 2 – 0: цвет символа.

Цвета кодируются в битах, как показано в таблице №2.3. .

Таблица №2.3. Атрибуты символов

	Обычный цвет	**Яркий цвет**
0 00b	**черный**	**темно-серый**
0 01b	**синий**	**светло-синий**
0 10b	**зеленый**	**светло-зеленый**
0 11b	**голубой**	**светло-голубой**
1 00b	**красный**	**светло-красный**
1 01b	**пурпурный**	**светло-пурпурный**
1 10b	**коричневый**	**желтый**
1 11b	**светло-серый**	**белый**

2.3.1 Считываем символ и атрибут символа в текущей позиции курсора

INT 10h, **AH** = 08 — Считать символ и атрибут символа в текущей позиции курсора

Ввод:	**AH** = 08 **BH** = номер страницы
Вывод:	**AH** = атрибут символа **AL** = ASCII-код символа

2.3.1 Выводим символ с заданным атрибутом на экран

INT 10h, AH = 09 — Вывести символ с заданным атрибутом на экран

Ввод:	**AH** = 09 **BH** = номер страницы **AL** = ASCII-код символа **BL** = атрибут символа **CX** = число повторений символа

С помощью этой функции можно вывести на экран любой символ, включая даже символы CR и LF, которые обычно интерпретируются как конец строки. В графических режимах CX не должен превышать число позиций, оставшееся до правого края экрана.

Пример работы.

Пример № 2.2.1

1.	**.model tiny**	; модель памяти, в которой сегменты кода, данных и стека объединены.
2.	**.code**	; сегмент кода, который содержит данные.
3.	**org 100h**	; начало COM-файла
4.	**begin:**	; метка начала кода программы
5.	**mov ah,09**	; помещаем номер функции DOS «вывод строки (9)» в регистр AH.
6.	**mov bh,0**	; устанавливаем номер страницы
7.	**mov al, < ASCII-код символа >**	; строка 12; заносим в регистр **al** – любой ASCII-код символа
8.	**mov bl, 00011111b**	; атрибут символа (белый на голубом)
9.	**mov cx,555**	; устанавливаем в счетчик кол-во выводимых символов
10.	**int 10h**	; вызов прерывания DOS – вызов символа;
11.	**ret**	; функция DOS "завершить программу"
12.	**end begin**	; метка окончания кода программы

2.3.2 Выводим символ с текущим атрибутом на экран

INT 10h, AH = 0Ah — Вывести

Ввод:	**AH** = 0Ah **BH** = номер страницы **AL** = ASCII-код символа **CX** = число повторений символа

Эта функция также выводит любой символ на экран, но в качестве атрибута символа используется атрибут, который имел символ, находившийся ранее в этой позиции.

2.3.3 Выводим символ в режиме телетайпа

INT 10h, AH = 0Eh — Вывести символ в режиме телетайпа

Ввод:	**AH** = **0Eh** **BH** = номер страницы **AL** = ASCII-код символа

Символы CR (0Dh), LF (0Ah), BEL (7) интерпретируются как управляющие символы. Если текст при записи выходит за пределы нижней строки, экран прокручивается вверх. В качестве атрибута используется атрибут символа, находившегося в этой позиции.

2.3.4 Выводим строку символов с заданными атрибутами

INT 10h, **AH = 13h** — Вывести строку символов с заданными атрибутами

Ввод:	**AH** = 13h **AL** = режим вывода:

	Бит 0 — переместить курсор в конец строки после вывода бит 1 — строка содержит не только символы, но также и атрибуты, так что каждый символ описывается двумя байтами: ASCII-код и атрибут биты 2 – 7 зарезервированы **CX** = длина строки (только число символов) **BL** = атрибут, если строка содержит только символы **DH,DL** = строка и столбец, начиная с которых будет выводиться строки **ES:BP** = адрес начала строки в памяти

Функция **13h** выводит на экран строку символов, интерпретируя управляющие символы CR (**0Dh**), LF (**0Ah**), BS (**08**) и BEL (**07**). Если строка подготовлена в формате символ, атрибут — гораздо быстрее просто скопировать ее в видеопамять.

Функции BIOS удобны для переключения и настройки видеорежимов, но часто оказывается, что вывод текста на экран гораздо быстрее и проще выполнять просто копированием изображения в видеопамять.

3. Прямая работа с видеопамятью

Все, что изображено на мониторе — и графика, и текст, одновременно присутствует в памяти, встроенной в видеоадаптер. Для того чтобы изображение появилось на мониторе, оно должно быть записано в память видеоадаптера. Для этого отводится специальная область памяти, начинающаяся с абсолютного адреса 0B800h:0000h (для текстовых режимов) и заканчивающаяся на 0B800h:FFFFh. Все, что программы пишут в эту область памяти, немедленно пересылается в память видеоадаптера. В текстовых режимах для хранения каждого изображенного символа используются два байта: байт с ASCII-кодом символа и байт с его атрибутом, так что по адресу 0B800h:0000h лежит байт с кодом символа, находящимся в верхнем левом углу экрана; по адресу 0B800h:0001h лежит атрибут этого символа; по адресу 0B800h:0002h лежит код второго символа в верхней строке экрана и т.д.

Таким образом, любая программа может вывести текст на экран, простой командой пересылки данных, не прибегая ни к каким специальным функциям DOS или BIOS.

*Пример работ с видеопамятью***.**

Пример № 3.1

1.	**.model tiny**	; модель памяти, в которой сегменты кода, данных и стека объединены.
2.	**.code**	; сегмент кода, который содержит данные.
3.	**org 100h**	; начало COM-файла
4.	**begin:**	; метка начала кода программы
5.	**mov ax,0003h**	; видеорежим 3 (очистка экрана)
6.	**int 10h**	; прерывание DOS – очистка экрана;
7.	**mov ax,0B800h**	;загружаем в сегментный регистр **ES** число 0B800h
8.	**mov es,ax**	
9.	**mov di,0**	;загружаем в регистр **DI** нуль
10.	**mov ah,31**	; заносим в регистр **ah** - атрибут символа
11.	**mov al, < ASCII-код символа >**	заносим в регистр **al** – любой ASCII-код символа
12.	**mov es:[di],ax**	; заносим по адресу 0B800:0000h атрибут и ASCII-код символа
13.	**mov ah,10h**	; вызываем функцию 10h - чтобы можно было остановить программу до нажатия любой

		клавиши
14.	**int 16h**	; вызываем прерывание 16h - сервис работы с клавиатурой BIOS
15.	**ret**	; функция DOS "завершить программу"
16.	**end begin**	; метка окончания кода программы

При подготовке данных для копирования в видеопамять в этой программе в строках (7) и (8) загружаем в сегментный регистр **ES** число 0B800h, которое соответствует сегменту дисплея в текстовом режиме. В строке (9) загружаем в регистр DI нуль. Это будет смещение относительно сегмента 0B800h. В строках (10) и (11) в регистр **AH** заносится атрибут символа (31 - ярко-белый символ на синем фоне) и в **AL** - ASCII-код символа (01 - рожица) соответственно.

В строке (12) заносим по адресу 0B800:0000h (т.е. первый символ в первой строке дисплея - верхний левый угол) атрибут и ASCII-код символа (31 и 01 соответственно).

4. Задание для выполнения.

4.1. С помощью редактора эмулятора EMU 8086 напишите программы примеры, которых приведены в данной лабораторной работе.

4.2 Создайте файлы типа **MZ** и *.com.

4.3 Изучите структуру откомпилированных программ.

4.4 Получите задание у преподавателя (один из пяти вариантов табл.№1) напишите программу вывода на экран строки '**Hello**'.

4.5 Напишите программу работы переключения SuperVGA-видеорежимов (согласно вариантов табл. №2)

4.6. Подготовьте отчет, который должен содержать тексты программ, а также укажите описание работы команд программ.

4.7. В отчете должны содержаться ответы на следующие вопросы.

5. Контрольные вопросы

5.1 Перечислите функции вывода на экран средствами операционной системы DOS?

5.2 Принцип работы функции DOS **02h**?

5.3 Укажите основные управляющие символы вывода на экран?

5.4 Каким образом осуществить вывод программы в файл?

5.5 Укажите отличие функции DOS **02h** от **06h**?

5.6 Прерывание **int 29h**. Преимущества использования?

5.7 С помощью каких функций можно установить нужный видеорежим (текстовый, цветной, монохромный)?

5.8 Отметьте основные моменты установки super VGA-видеорежимов?

5.9 Укажите функции и прерывания управления положением курсора.

5.10 Перечислите функции считывания положения и размера курсора?

5.11 Вывод символов на экран средствами BIOS. Функции?

5.12 Прямая работа с видеопамятью. Принципы работы с видеопамятью?

5.13 Укажите преимущества вывода на экран с помощью непосредственной работы с видеопамятью?

5.14 Область памяти видеоадаптера?

5.15 Укажите код третьего символа в верхней строке экрана для работы с видеопамятью?

5.16 Если в примере № 1.2 длину строку указать большую, чем указанная что в данном случае будет выводиться на экрана

Табл. №1

№ вар.	Функция вывода (DOS)	Функция вывода (BIOS)	Видеопамять
6.	**02h**	**Ah=02h**	**‘Hello’**
7.	**06h**	**Ah=08h**	
8.	**09h**	**Ah=09h**	
9.	**40h**	**Ah=0Ah**	
10.	**29h**	**Ah=13h**	

Примечание: В примерах, в которых возможно задание различных параметров вывода (цвет символа, фона; номер строки, столбца, страницы и т.д.) выводите на экран слово «hello» с параметрами отличными от стандартных.

Табл. №2

№ вар.	*SuperVGA-видеорежим*
1.	**108h**
2.	**109h**
3.	**10Ah**
4.	**10Bh**
5.	**10Ch**

ЛАБОРАТОРНАЯ РАБОТА №7 - 8

КОМАНДЫ ЛОГИЧЕСКИХ ОПЕРАЦИЙ

Цель работы: *ознакомиться с работой команд логических операций: and, or, xor, test, not и реализацией их работы на практике*.

5. Краткие теоретические сведения.

Логические операции являются важным элементом в проектировании микросхем и имеют много общего в логике программирования. Команды **AND, OR, XOR** и **TEST** - являются командами логических операций. Эти команды используются для сброса и установки бит и для арифметических операций в коде ASCII. Все эти команды обрабатывают один байт или одно слово в регистре или в памяти, и устанавливают флаги **CF, OF, PF, SF, ZF.**

1. Команда AND

Команда **AND** (Логическое И) осуществляет логическое (побитовое) умножение первого операнда на второй. Исходное значение первого операнда (приемника) теряется, замещаясь результатом умножения. В качестве первого операнда команды and можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги **SF**, **ZF** и **PF**.

Правила побитового умножения:

Первый операнд-бит 0101	**Бит результата 0001**
Второй операнд-бит 0011	

Пример 1

```
mov AX,0FFEh
and AX,5555h          ;AX=0554h
```

Пример 2

```
mov ax,00101001b
add ax,11110111b      ; ax=00100001b
```

2. Команда OR

Команда **OR (**Логическое ВКЛЮЧАЮЩЕЕ ИЛИ**)** выполняет операцию логического (побитового) сложения двух операндов. Результат замещает первый операнд (приемник); второй операнд (источник) не изменяется. В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды команды **OR** могут быть байтами или словами. Команда воздействует на флаги **OF, SF, ZF, PF** и **CF**, при этом флаги **CF** и **OF** всегда сбрасываются в 0.

Правила побитового сложения:

Первый операнд-бит 0101	**Бит результата 0111**
Второй операнд-бит 0011	

Пример 1

```
mov AX,000Fh
mov BX,00F0h
or AX,BX                ;AX=00FFh, BX=00F0h
```

Пример 2

```
mov AX,00101001b
mov BX,11110111b
or AX,BX                ;mov dx,11111111b
```

Пример 3

```
mov AX,000Fh
or AX,8001h             ;AX=800Fh
```

3. Команда XOR.

Команда **XOR** (Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ) выполняет операцию логического (побитового) ИСКЛЮЧАЮЩЕГО ИЛИ над своими двумя операндами. Результат операции замещает первый операнд; второй операнд не изменяется. Каждый бит результата устанавливается в 1, если соответствующие биты операндов различны, и сбрасывается в 0, если соответствующие биты операндов совпадают.

В качестве первого операнда команды **XOR** можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги **OF, SF, ZF, PF** и **CF**, причем флаги **OF** и **CF** всегда сбрасываются, а остальные флаги устанавливаются в зависимости от результата.

Правила побитового ИСКЛЮЧАЮЩЕГО ИЛИ:

Первый операнд-бит 0101	**Бит результата 0110**
Второй операнд-бит 0011	

Пример 1

```
mov AX,0Fh
xor AX,0FFFFh ;AX=FFF0h
```

Пример 2

```
mov AX,00101001b
mov BX,11110111b
xor ax,bx               ; 11011110b
```

Пример 3

```
mov SI,0AAAAh
mov BX,5555h
xor SI,BX               ;SI=FFFFh,BX=5555h
```

Пример 4

```
xor BX,BX               ;Обнуление BX
```

4. Команда TEST.

Команда **TEST** (Логическое сравнение) выполняет операцию логического умножения И над двумя операндами и, в зависимости от результата, устанавливает флаги **SF, ZF** и **PF**. Флаги **OF** и **CF** сбрасываются, а **AF** имеет неопределенное значение. Состояние флагов можно затем проанализировать командами условных переходов. *Команда **TEST** не изменяет ни один из операндов.*

В качестве первого операнда команды **TEST** можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго - регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

Правила побитового умножения:

Первый операнд-бит 0101	**Бит результата 0001**
Второй операнд-бит 0011	

Флаг **SF** устанавливается в 1, если в результате выполнения команды образовалось число с установленным знаковым битом.

Флаг **ZF** устанавливается в 1, если в результате выполнения команды образовалось число, состоящее из одних двоичных нулей.

Флаг **PF** устанавливается в 1, если в результате выполнения команды образовалось число с четным количеством двоичных единиц в его битах.

Пример 1
test AX,1
jne label2: ;Переход, если бит 0 в AX установлен
je label1: ;Переход, если бит 0 в AX сброшен

Пример №1.4.1.1

1.	**.model tiny**	; модель памяти в которой сегменты кода, данных и стека объединены.
1	**.code**	; сегмент кода, который содержит данные.
2	**org 100h**	; начало COM-файла
3	**begin:**	; метка начала кода программы
4	**mov CX,<число1 >**	; загружаем в **CX** число1 <любое число1>
5	**mov BX,<число2>**	; загружаем в **BX** число2 <любое число2>
6	**test cx,bx**	; логически сравниваем числа в регистрах cx с bx
7	**jne label2**	; если одно из значений не равно 0 то переходим на метку **label2**
8	**je label1**	; если одно из значений равно 0 то переходим на метку **label1**
9	**ret**	; функция DOS "завершить программу" (не выполняется)
10	**label1:**	; начало блока метки **Label1**
11	**mov ah,9**	; помещаем номер функции DOS «вывод строки (9)» в регистр AH.

12	**mov dx,offset string**	помещает в регистр DX смещение метки **String** относительно начала сегмента данных
13	**int 21h**	; функция DOS "вывод строки"
14	**ret**	; функция DOS "завершить программу"
15	**String db 'одно из чисел равно 0$'**	**;** строка с содержащая выводимые данные.
16	**label2:**	; начало блока метки **Label2**
17	**mov ah,9**	; помещаем номер функции DOS «вывод строки (9)» в регистр AH.
18	**mov dx,offset string1**	помещает в регистр DX смещение метки **String1** относительно начала сегмента данных
19	**int 21h**	; функция DOS "вывод строки"
20	**ret**	; функция DOS "завершить программу"
21	**string1 db 'не равны 0$'**	**;** строка с содержащая выводимые данные.
22	**end begin**	; метка окончания кода программы

Данный пример сравнивает два значения (строка (6)), если одно из двух значений равно нулю тогда переходим на метку **label1** (строка(10)) далее выполняются команды, следующие после этой метки, в случае если одно из двух значений равно нулю тогда переходим на метку **label2**

Пример 2

```
test SI,8
jne bityes          ;Переход, если бит 3 в SI установлен
je bitno            ;Переход, если бит 0 в AX сброшен
```

Пример 3

```
test DX,0FFFFh
jz null                 ;Переход, если DX=0
jnz smth            ;Переход, если DX не 0
```

5. Команда NOT.

Команда **NOT (**NOT Инверсия, дополнение до 1, логическое отрицание) выполняет инверсию битов указанного операнда, заменяя 0 на 1 и наоборот. В качестве операнда можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Команда не воздействует на флаги процессора.

Правила побитовой инверсии:

Операнд-бит 0 1	Бит результата 1 0

Пример 1

```
mov AX,0FFFFh
not AX ;AX=0000h
```

Пример 2
mov SI,5551h
not SI ;SI=AAAEh

6. Характерные примеры работы команд логических операций.

Для следующих несвязанных примеров, предположим, что:
- **AL** содержит 1100 0101
- **BH** содержит 0101 1100:

1. **AND AL,BH** ;Устанавливает в **AL** 0100 0100
2. **OR BH,AL** ;Устанавливает в **BH** 1101 1101
3. **XOR AL,AL** ;Устанавливает в **AL** 0000 0000
4. **AND AL,00** ;Устанавливает в **AL** 0000 0000
5. **AND AL,0FH** ;Устанавливает в **AL** 0000 0101
6. **OR CL,CL** ;Устанавливает флаги **SF** и **ZF**

Примеры 3 и 4 демонстрируют способ очистки регистра.
В примере 5 обнуляются левые четыре бита регистра AL.
Можно применить команду **OR** для следующих целей:

1. **OR CX,CX** ;Проверка CX на нуль
   JZ ;Переход, если нуль
2. **OR CX,CX** ;Проверка знака в CX
   JS ;Переход, если отрицательно

3. Задание для выполнения.

3.1 Запустить эмулятор EMU8086.
3.2 Пользуясь правилами оформления ассемблерных программ, наберите код примера **№1.4.1.1**, запустите код на выполнение.
3.3 Проанализируйте работу кода примера **№1.4.1.1**
3.4 С помощью справки эмулятора EMU8086 выясните работу команд (**jne, je, js, jz**)
3.5 Получите задание у преподавателя (один из четырех вариантов (команды(**and, or, xor, test, not**))) и, пользуясь правилами оформления ассемблерных программ, напишите три программы характеризующие (показывающие) работу их с числами (двоичной, десятеричной, шестнадцатеричной систем счисления) согласно перечислению приведенных примеров**.**
3.6 Результаты работы продемонстрируйте преподавателю.

4. Контрольные вопросы

4.1 Назначение команд логических операций?
4.2 Команда **and** основное назначение?
4.3 Команда **or** основное назначение?
4.4 Команда **xor** основное назначение?
4.5 Команда **test** основное назначение?
4.6 Команда **not** основное назначение?
4.7 Альтернативная работа команд (**test, xor, and**)?
4.8 Допустим что будит

4.9 В чем заключается работа связки команд **jne, je?**
4.10 В чем заключается работа связки команд **js, jz?**
4.11 Что произойдет, если в примере **№1.4.1.1** мы изменим строку (8) на следующую **(jne label1)?**

ОЗНАКОМЛЕНИЕ С РАБОТОЙ ЦИКЛОВ

Цель работы: *ознакомиться со структурой и реализацией циклов в программе.*

1. Краткие теоретические сведения.

Циклы, позволяющие выполнить некоторый участок программы многократно, в любом языке являются одной из наиболее употребительных конструкций. В системе команд МП 86 циклы реализуются, главным образом, с помощью команды **loop** (петля), хотя имеются и другие способы организации циклов. В большинстве случаев число шагов в цикле определяется содержимым регистра **CX**, поэтому максимальное число шагов составляет 64 К.

Организация циклических переходов, как на языках высокого уровня, так и на языке assembler представляет собой замечательное средство, позволяющее значительно снизить код исполняемой программы.

В общем виде любой цикл записывается в ассемблере как условный переход.

1. Организация цикла с помощью команды LOOP (Первый способ).

Команда **loop** (анг. петля) выполняет декремент содержимого регистра **CX** (счетчик), и если оно не равно 0, осуществляет переход на указанную метку вперед или назад в том же программном сегменте в диапазоне -128... + 127 байт. Обычно метка помещается перед первым предложением тела цикла, а команда **loop** является последней командой цикла. Содержимое регистра **CX** рассматривается как целое число без знака, поэтому максимальное число повторений группы включенных в цикл команд составляет 65536 (если перед входом в цикл **CX=0**). Команда не воздействует на флаги процессора.

Команда	Назначение	Процессор
LOOP метка	Организация циклов	8086

*Простейший пример организации циклического перехода (со счетчиком в регистре **cx**) на языке Assembler:*

38.	**.model tiny**	; модель памяти, в которой сегменты кода, данных и стека объединены.
39.	**.code**	; сегмент кода, который содержит и данные.
40.	**org 100h**	; начало COM-файла
41.	**begin:**	; метка начала кода программы
42.	**mov cx,10**	; загружаем в (регистр-счетчик) **CX** количество повторов (отсчет будет идти от 10 до 0)
43.	**Label1:**	; создаем метку (Label - метка).
44.	**mov ah,9**	; помещаем номер функции DOS «вывод строки (9)» в регистр AH.
45.	**mov dx,offset String**	помещает в регистр **DX** смещение метки **String** относительно начала сегмента данных
46.	**int 21h**	; функция DOS "вывод строки"

47.	**loop Label1**	; оператор **loop** уменьшает на единицу **CX** и, если он не равен нулю, переходит на метку Label1 (строка 6)
48.	**ret**	; функция DOS "завершить программу"
49.	**string db 'privet $'**	**;** строка с содержащая выводимые данные.
50.	**end begin**	; метка окончания кода программы

В строке (5) загружаем в **CX** количество повторов (отсчет будет идти от 10 до 0). В строке (6) создаем метку (Label - метка). Далее (строки (7)-(9)) выводим сообщение. И в строке (10) оператор loop уменьшает на единицу **CX** и, если он не равен нулю, переходит на метку Label1 (строка (6)). Таким образом, строка будет выведена на экран десять раз. Когда программа перейдет на строку (11), регистр **CX** будет равен нулю.

2. Организация цикла с помощью команды JMP (Второй способ).

Команда **jmp** передает управление в указанную точку того же или другого программного сегмента. Адрес возврата не сохраняется. Команда не воздействует на флаги процессора.

Команда **jmp** имеет пять разновидностей:

- переход прямой короткий (в пределах -128... + 127 байтов);
- переход прямой ближний (в пределах текущего программного сегмента);
- переход прямой дальний (в другой программный сегмент);
- переход косвенный ближний;
- переход косвенный дальний.

Все разновидности переходов имеют одну и ту же мнемонику **jmp**, хотя и различающиеся коды операций. Во многих случаях транслятор может определить вид перехода по контексту, в тех же случаях, когда это невозможно, следует использовать атрибутные операторы (**short** - прямой короткий переход; **near ptr** - прямой ближний переход; **far ptr** - прямой дальний переход; **word ptr** - косвенный ближний переход; **dword ptr** - косвенный дальний переход).

Команда	Назначение	Процессор
JMP метка	Безусловный переход	8086

1.	**.model tiny**	;модель памяти, в которой сегменты кода, данных и стека объединены.
2.	**.code**	;сегмент кода, который содержит данные.
3.	**org 100h**	; начало COM-файла
4.	**begin:**	;метка начала кода программы
5.	**label1:**	;создаем метку
6.	**mov ah,9**	;помещаем номер функции DOS «вывод строки (9)» в регистр **AH**.
7.	**mov dx,offset String**	помещает в регистр **DX** смещение метки **String** относительно начала сегмента данных
8.	**int 21h**	;функция DOS "вывод строки"
9.	**jmp Label1**	; переход на строку с меткой **Label1**
10.	**add cx,12**	;прибавить к значению регистра **cx** число 12 (данная команда не выполняется)
11.	**dec cx**	;уменьшить значение регистра **cx** на 1

		(данная команда не выполняется)
12.	**ret**	;функция DOS "завершить программу"
13.	**string db "PRIVET",13,10,'$'**	**;**строка с содержащая выводимые данные.
14.	**end begin**	; метка окончания кода программы

В результате работы программы будет зациклен блок строк (6) - (10) (Вывод строки **PRIVET** многочисленное количество раз) Строки (10)-(11).

22.3 Организация цикла с помощью команд DEC и JNZ (Третий способ).

С помощь этих операторов можно создавать циклы, которые будут работать быстрее оператора **Loop**. Комбинированная работа команд **DEC** и **JNZ** уменьшает содержимое регистра **CX** на 1 и выполняет переход на метку, если в **CX** не равен нулю.

Команда **DEC,** кроме того, устанавливает флаг нуля во флаговом регистре в состояние 0 или 1. Команда **JNZ** затем проверяет эту установку.

Аналогично командам **JMP** и **LOOP** операнд в команде **JNZ** содержит значение расстояния между концом команды **JNZ** и адресом перехода(**Label1**), которое прибавляется к командному указателю. Это расстояние должно быть в пределах от -128 до +127 байт.

Следующий пример будет работать так же, как и Пример №1.1, только быстрее.

1.	**.model tiny**	; модель памяти, в которой сегменты кода, данных и стека объединены.
23	**.code**	; сегмент кода, который содержит данные.
24	**org 100h**	; начало COM-файла
25	**begin:**	; метка начала кода программы
26	**mov cx,10**	; загружаем в (регистр-счетчик) **CX** количество повторов (отсчет будет идти от 10 до 0)
27	**Label1:**	; создаем метку (Label - метка).
28	**mov ah,9**	; помещаем номер функции DOS «вывод строки (9)» в регистр **AH**.
29	**mov dx,offset String**	помещает в регистр **DX** смещение метки **String** относительно начала сегмента данных
30	**int 21h**	; функция DOS "вывод строки"
31	**dec cx**	; оператор **DEC** уменьшает на единицу **CX** и, если он не равен нулю, переходит на метку **Label1**
32	**jnz Label1**	**;** условный переход на строку с меткой **Label1**
33	**ret**	; функция DOS "завершить программу"
34	**string db 'priver ',13,10, '$'**	**;** строка с содержащая выводимые данные
35	**end begin**	; метка окончания кода программы

2.Программа для практики.

Напишем программу, выводящую на экран все ASCII-символы (16 строк по 16 символов в строке).

1.	**.model tiny**	; модель памяти в которой сегменты кода, данных и стека объединены.
36	**.code**	; сегмент кода, который содержит данные.
37	**org 100h**	; начало COM-файла
38	**begin:**	; метка начала кода программы
39	**mov cx,256**	; задаем значение счетчика (256 символов)
40	**mov dl,0**	; первый символ - с кодом 00
41	**mov ah,2**	; номер функции DOS "вывод символа"
42	**cloop: int 21h**	; вызов DOS
43	**inc dl**	; увеличение DL на 1 - следующий символ
44	**test dl,0Fh**	; если DL не кратен 16
45	**jnz continue_loop;**	;продолжить цикл,
46	**push dx**	; иначе: сохранить текущий символ
47	**mov dl,0Dh**	; вывести CR
48	**int 21h**	; вызов DOS
49	**mov dl,0Ah**	; вывести LF
50	**int 21h**	; вызов DOS
51	**pop dx**	; восстановить текущий символ
52	**continue_loop:**	; метка
53	**loop cloop**	; продолжить цикл
54	**ret**	; завершение COM-файла
55	**end begin**	; метка окончания кода программы

Здесь с помощью команды **LOOP** оформляется цикл, выполняющийся 256 раз (значение регистра **CX** в начале цикла). Регистр **DL** содержит код символа, который равен нулю в начале цикла и увеличивается каждый раз на 1 командой **INC DL**. Если значение **DL** сразу после увеличения на 1 кратно 16, оно временно сохраняется в стеке и на экран выводятся символы **CR** и **LF**, выполняющие переход на начало новой строки. Проверка выполняется командой **TEST DL,0Fh** — результат операции **AND** над **DL** и **0Fh** будет нулем, только если младшие четыре бита **DL** равны нулю, что и соответствует кратности шестнадцати.

3. Содержание отчета.

3.1. Титульный лист.

3.2. Индивидуальный вариант задания.

3.3. Тестовые наборы данных и предполагаемые результаты.

3.4. Текст программы до отладки.

3.5. Список ошибок, обнаруженных при отладке.

3.6. Результаты выполнения тестов.

3.7. Распечатка листинга компиляции отлаженной программы с указанием работы каждой строки.

4. Задание для выполнения.

4.1. Выполните все примеры, что содержатся в описании данной лабораторной работы.

4.2. Проанализируйте работу программы примера для практики.

4.3 Изучить условия организации циклических переходов на языке Ассемблера.

4.4. Напишите программу, выводящую на экран слово **«!!!!!!!!!! Hello !!!!!!!!!!»** используя команды циклических переходов (3 варианта).

4.5. Получите задание у преподавателя (один из пяти вариантов табл.№1) и, пользуясь правилами оформления ассемблерных программ, создайте программу, выводящую на экран слово, **D** число раз.

4.6 Программу ассемблируйте в файл типа ***.com** или ***.exe** (на выбор);

5. Контрольные вопросы

5.1 Организация цикла с помощью команды **loop**?

5.2 Значимость регистра **cx**?

5.3 Максимальное число повторений команд цикла определяемого регистром **cx** ?

5.4 Организация цикла с помощью команды **jmp**?

5.5. Разновидности команды **jmp**?

5.6. Организация цикла с помощью команд **dec и jnz**?

Табл. №1

№ вар.	**Выводимые данные**	**Формула расчета**	**A**	**B**	**C**
1.	Циклический переход	**D=A+B+C**	**101**	**345**	**121**
2.	Hello world	**D=A-B+C**	**578**	**152**	**149**
3.	Good Bye	**D=A+B-C**	**333**	**223**	**16**
4.	Группа	**D=A-B+C**	**1502**	**834**	**1**
5.	Лабораторная работа	**D=A-B-C**	**1056**	**33**	**125**

ЛАБОРАТОРНАЯ РАБОТА №9 - 10

РАБОТА С СИМВОЛЬНИМИ СТРОКАМИ

1. Цель работы

Закрепление практических навыков в работе с массивами и указателями языка С, обеспечении функциональной модульности

2. Темы для предварительного изучения

- Указатели в языке С.
- Представление строк.
- Функции и передача параметров.

3. Постановка задачи

По индивидуальному заданию создать функцию для обработки символьных строк. За образец брать библиотечные функции обработки строк языка С, но не применять их в своей функции. Предусмотреть обработку ошибок в задании параметров и особые случаи. Разработать два варианта заданной функции - используя традиционную обработку массивов и используя адресную арифметику.

4. Индивидуальные задания

╪п.п	**Вариант**
1	Функция - Copies(s,s1,n). Назначение - копирование строки s в строку s1 n раз
2	Функция - Words(s). Назначение - подсчет слов в строке s
3	Функция - Concat(s1,s2). Назначение - конкатенация строк s1 и s2 (аналогичная библиотечная функция С - strcat)
4	Функция - Parse(s,t). Назначение - разделение строки s на две части: до первого вхождения символа t и после него
5	Функция - Center(s1,s2,l). Назначение - центрирование - размещение строки s1 в середине строки s2 длиной l
6	Функция - Delete(s,n,l). Назначение - удаление из строки s подстроки, начиная с позиции n, длиной l (аналогичная библиотечная Функция есть в Pascal).
7	Функция - Left(s,l). Назначение - выравнивание строки s по левому краю до длины l.
8	Функция - Right(s,l) Назначение - выравнивание строки s по правому краю до длины l.
9	Функция - Insert(s,s1,n). Назначение - вставка в строку s подстроки s1, начиная с позиции n (аналогичная библиотечная функция есть в Pascal).
10	Функция - Reverse(s). Назначение - изменение порядка символов в строке s на противоположный.
11	Функция - Pos(s,s1). Назначение - поиск первого вхождения подстроки s1 в строку s (аналогичная функция С - strstr).
12	Функция - LastPos(s,s1). Назначение - поиск последнего вхождения подстроки s1 в строку s.
13	Функция - WordIndex(s,n). Назначение - определение позиции начала в строке s слова с номером n.
14	Функция - WordLength(s,n). Назначение - определение длины слова с номером n в строке s.

15 Функция - SubWord(s,n,l). Назначение - выделение из строки s l слов, начиная со слова с номером n.

16 Функция - WordCmp(s1,s2). Назначение - сравнение строк (с игнорированием множественных пробелов).

17 Функция - StrSpn(s,s1). Назначение - определение длины той части строки s, которая содержит только символы из строки s1.

18 Функция - StrCSpn(s,s1). Назначение - определение длины той части строки s, которая не содержит символы из строки s1.

19 Функция - Overlay(s,s1,n). Назначение - перекрытие части строки s, начиная с позиции n, строкой s1.

20 Функция - Replace(s,s1,s2). Назначение - замена в строке s комбинации символов s1 на s2.

21 Функция - Compress(s,t). Назначение - замена в строке s множественных вхождений символа t на одно.

22 Функция - Trim(s). Назначение - удаление начальных и конечных пробелов в строке s.

23 Функция - StrSet(s,n,l,t). Назначение - установка l символов строки s, начиная с позиции n, в значение t.

23 Функция - Space(s,l). Назначение - доведение строки s до длины l путем вставки пробелов между словами.

24 Функция - Findwords(s,s1). Назначение - поиск вхождения в строку s заданной фразы (последовательности слов) s1.

25 Функция - StrType(s). Назначение - определение типа строки s (возможные типы - строка букв, десятичное число, 16-ричное число, двоичное число и т.д.).

26 Функция - Compul(s1,s2). Назначение - сравнение строк s1 и та s2 с игнорированием различий в регистрах.

27 Функция - Translate(s,s1,s2). Назначение - перевод в строке s символов, которые входят в алфавит s1, в символы, которые входят в алфавит s2.

28 Функция - Word(s). Назначение - выделение первого слова из строки s.

Примечание: под "словом" везде понимается последовательность символов, которая не содержит пробелов.

5. Пример решения задачи

5.1. Индивидуальное задание:

- Функция - substr(s,n,l). Назначение - выделение из строки s подстроки, начиная с позиции n, длиной l.

5.2. Описание метода решения

5.2.1. Символьная строка в языке С представляется в памяти как массив символов, в конце которого находится байт с кодом 0 - признак конца строки. Строку, как и любой другой массив можно обрабатывать либо традиционным методом - как массив, с использованием операции индексации, либо через указатели, с использованием операций адресной арифметики. При работе со строкой как с массивом нужно иметь в виду, что длина строки заранее неизвестна, так что циклы должны быть организованы не со счетчиком, а до появления признака конца строки.

5.2.2. Функция должна реализовывать поставленную задачу - и ничего более. Это означает, что функцию можно будет, например, перенести без изменений в любую другую программу,

если спецификации функции удовлетворяют условиям задачи. Это также означает, что при ошибочном задании параметров или при каких-то особых случаях в их значениях функция не должна аварийно завершать программу или выводить какие-то сообщения на экран, но должна возвращать какое-то прогнозируемое значение, по которому та функция, которая вызвала нашу, может сделать вывод об ошибке или об особом случае.

Определим состав параметров функции:

int substr (src, dest, num, len);

где src - строка, с которой выбираются символы;

dest - строка, в которую записываются символы;

num - номер первого символа в строке src, с которого начинается подстрока (нумерация символов ведется с 0);

len - длина выходной строки.

Возможные возвращаемые значения функции установим: 1 (задание параметров правильное) и 0 (задание не правильное). Эти значения при обращениях к функции можно будет интерпретировать как "истина" или "ложь".

Обозначим через Lsrc длину строки src. Тогда возможны такие варианты при задании параметров:

- num+len <= Lsrc - полностью правильное задание;
- num+len > Lsrc; num < Lsrc - правильное задание, но длина выходной строки буде меньше, чем len;
- num >= Lsrc - неправильное задание, выходная строка будет пустой;
- num < 0 или len <= 0 - неправильное задание, выходная строка будет пустой.

Заметим, что интерпретация конфигурации параметров как правильная / неправильная и выбор реакции на неправильное задание - дело исполнителя. Но исполнитель должен строго выполнять принятые правила. Возможен также случай, когда выходная строка выйдет большей длины, чем для нее отведено места в памяти. Однако, поскольку нашей функции неизвестен размер памяти, отведенный для строки, функция не может распознать и обработать этот случай - так же ведут себя и библиотечные функции языка С.

5.3. Описание логической структуры

5.3.1. Программа состоит из одного программного модуля - файл LAB1.C. В состав модуля входят три функции - main, substr_mas и subs_ptr. Общих переменных в программе нет. Макроконстантой N определена максимальная длина строки - 80.

5.3.2. Функция main является главной функцией программы, она предназначена для ввода исходных данных, вызова других функций и вывода результатов. В функции определены переменные:

- ss и dd - входная и выходная строки соответственно;
- n - номер символа, с которого должна начинаться выходная строка;
- l - длина выходной строки.

Функция запрашивает и вводит значение входной строки, номера символа и длины. Далее функция вызывает функцию substr_mas, передавая ей как параметры введенные

значения. Если функция substr_mas возвращает 1, выводится на экран входная и выходная строки, если 0 - выводится сообщение об ошибке и входная строка. Потом входная строка делается пустой и то же самое выполняется для функции substr_ptr.

5.3.3. Функция substr_mas выполняет поставленное задание методом массивов. Ее параметры: - src и dest - входная и выходная строки соответственно, представленные в виде массивов неопределенного размера; num и len. Внутренние переменные i и j используются как индексы в массивах.

Функция проверяет значения параметров в соответствии со случаем 4, если условия этого случая обнаружены, в первый элемент массива dest записывается признак конца строки и функция возвращает 0.

Если случай 4 не выявлен, функция просматривает num первых символов входной строки. Если при этом будет найден признак конца строки, это - случай 3, при этом в первый элемент массива dest записывается признак конца строки и функция возвращает 0.

Если признак конца в первых num символах не найден, выполняется цикл, в котором индекс входного массива начинает меняться от 1, а индекс выходного - от 0. В каждой итерации этого цикла один элемент входного массива пересылается в выходной. Если пересланный элемент является признаком конца строки (случай 2), то функция немедленно заканчивается, возвращая 1. Если в цикле не встретится конец строки, цикл завершится после len итераций. В этом случае в конец выходной строки записывается признак конца и Функция возвращает 1.

5.3.4. Функция substr_ptr выполняет поставленное задание методом указателей. Ее параметры: - src и dest - входная и выходная строки соответственно, представленные в виде указателей на начала строк; num и len.

Функция проверяет значения параметров в соответствии со случаем 4, если условия этого случая выявлены, по адресу, который задает dest, записывается признак конца строки и функция возвращает 0, эти действия выполняются одним оператором.

Если случай 4 не обнаружен, функция пропускает num первых символов входной строки. Это сделано циклом while, условием выхода из которого является уменьшение счетчика num до 0 или появление признака конца входной строки. Важно четко представлять порядок операций, которые выполняются в этом цикле:

- выбирается счетчик num;
- счетчик num уменьшается на 1;
- если выбранное значение счетчика было 0 - цикл завершается;
- если выбранное значение было не 0 - выбирается символ, на который указывает указатель src;
- указатель src увеличивается на 1;
- если выбранное значение символа было 0, т.е., признак конца строки, цикл завершается, иначе - повторяется.

После выхода из цикла проверяется значение счетчика num: если оно не 0, это означает, что выход из цикла произошел по признаку конца строки (случай 3), по адресу, который задает dest, записывается признак конца строки и функция возвращает 0.

Если признак конца не найден, выполняется цикл, подобный первому циклу while, но по счетчику len. В каждой итерации этого цикла символ, на который показывает src переписывается по адресу, задаваемому dest, после чего оба указателя увеличиваются на 1. Цикл закончится, когда будет переписано len символов или встретится признак конца строки. В любом варианте завершения цикла по текущему адресу, который содержится в указателе dest, записывается признак конца строки и функция завершается, возвращая 1.

5.4. Данные для тестирования

Тестирование должно обеспечить проверку работоспособности функций для всех вариантов входных данных. Входные данные, на которых должно проводиться тестирование, сведены в таблицу:

вариант	src	num	len	dest
1	012345	2	2	23
	012345	0	1	0
	012345	0	6	012345
2	012345	5	3	5
	012345	2	6	2345
	012345	0	7	012345
3	012345	8	2	пусто
4	012345	-1	2	пусто
	012345	5	0	пусто
	012345	5	-1	пусто

5.5. Текст программы

```
/*****************************************************************/
/*********************  Файл LAB1.C **************************/
#include <stdio.h>
#define N 80
/*****************************************************************/
/*         Функция выделения подстроки (массивы)            */
/*****************************************************************/
int substr_mas(char src[N],char dest[N],int num,int len){
  int i, j;
  /* проверка случая 4 */
  if ( (num<0)||(len<=0) ) {
    dest[0]=0;  return 0;
    }
  /* выход на num-ый символ */
  for (i=0; i<=num; i++)
   /* проверка случая 3 */
   if ( src[i]=='\0') {
     dest[0]=0;  return 0;
     }
```

```
   /* перезапись символов */
      for (i--, j=0; j<len; j++, i++) {
     dest[j]=src[i];
     /* проверка случая 2 */
     if ( dest[j]=='\0') return 1;
     }
   /* запись признака конца в выходную строку */
   dest[j]='\0';
   return 1;
 }
/*****************************************************************/
/*       Функция выделение подстроки (адресная арифметика)     */
/*****************************************************************/
int substr_ptr(char *src, char *dest, int num, int len) {
   /* проверка случая 4 */
   if ( (num<0)||(len<=0) ) return dest[0]=0;
   /* выход на num-ый символ или на конец строки */
   while ( num-- && *src++ );
   /* проверка случая 3 */
   if ( !num ) return dest[0]=0;
   /* перезапись символов */
   while ( len-- && *src )  *dest++=*src++;
   /* запись признака конца в выходную строку */
   *dest=0;
   return 1;
 }
/*****************************************************************/
main()
  {
  char ss[N], dd[N];
  int n, l;
  clrscr();
  printf("Вводите строку:\n");
  gets(ss);
  printf("начало=");
  scanf("%d",&n);
  printf("длина=");
  scanf("%d",&l);
  printf("Массивы:\n");
  if (substr_mas(ss,dd,n,l))  printf(">>%s<<\n>>%s<<\n",ss,dd);
  else printf("Ошибка! >>%s<<\n",dd);
  dd[0]='\0';
  printf("Адресная арифметика:\n");
  if (substr_ptr(ss,dd,n,l))  printf(">>%s<<\n>>%s<<\n",ss,dd);
  else printf("Ошибка! >>%s<<\n",dd);
  getch();
  }
```

ЛАБОРАТОРНАЯ РАБОТА №11 - 12

ПРЕДСТАВЛЕНИЕ В ПАМЯТИ МАССИВОВ И МАТРИЦ

1. Цель работы

Получение практических навыков в использовании указателей и динамических объектов в языке С, создание модульных программ и обеспечение инкапсуляции.

2. Темы для предварительного изучения

- Указатели в языке С.
- Модульная структура программы.
- Области действия имен.

3. Постановка задачи

Для разряженной матрицы целых чисел в соответствии с индивидуальным заданием создать модуль доступа к ней, у котором обеспечить экономию памяти при размещении данных.

4. Порядок выполнения

Порядок выполнения работы и содержание отчета определены в общих указаниях.

5. Индивидуальные задания

╪п/п	Вид матрицы
1	все нулевые элементы размещены в левой части матрицы
2	все нулевые элементы размещены в правой части матрицы
3	все нулевые элементы размещены выше главной диагонали
4	все нулевые элементы размещены в верхней части матрицы
5	все нулевые элементы размещены в нижней части матрицы
6	все элементы нечетных строк - нулевые
7	все элементы четных строк - нулевые
8	все элементы нечетных столбцов - нулевые
9	все элементы четных столбцов - нулевые
10	все нулевые элементы размещены в шахматном порядке, начиная с 1-го элемента 1-й строки
11	все нулевые элементы размещены в шахматном порядке, начиная со-2го элемента 1-й строки
12	все нулевые элементы размещены на местах с четными индексами строк и столбцов
13	все нулевые элементы размещены на местах с нечетными индексами строк и столбцов
14	все нулевые элементы размещены выше главной диагонали на нечетных строках и ниже главной диагонали - на четных
15	все нулевые элементы размещены ниже главной диагонали на нечетных строках и выше главной диагонали - на четных
16	все нулевые элементы размещены на главной диагонали, в первых 3 строках выше диагонали и в последних 3 строках ниже диагонали
17	все нулевые элементы размещены на главной диагонали и в верхней половине участка

	выше диагонали
18	все нулевые элементы размещены на главной диагонали и в нижней половине участка ниже диагонали
19	все нулевые элементы размещены в верхней и нижней четвертях матрицы (главная и побочная диагонали делят матрицу на четверти)
20	все нулевые элементы размещены в левой и правой четвертях матрицы (главная и побочная диагонали делят матрицу на четверти)
21	все нулевые элементы размещены в левой и верхней четвертях матрицы (главная и побочная диагонали делят матрицу на четверти)
22	все нулевые элементы размещены на строках, индексы которых кратны 3
23	все нулевые элементы размещены на столбцах, индексы которых кратны 3
24	все нулевые элементы размещены на строках, индексы которых кратны 4
25	все нулевые элементы размещены на столбцах, индексы которых кратны 4
26	все нулевые элементы размещены попарно в шахматном порядке (сначала 2 нулевых)
27	матрица поделена диагоналями на 4 треугольники, элементы верхнего и нижнего треугольников нулевые
28	матрица поделена диагоналями на 4 треугольники, элементы левого и правого треугольников нулевые
29	матрица поделена диагоналями на 4 треугольника, элементы правого и нижнего треугольников нулевые
30	все нулевые элементы размещены квадратами 2x2 в шахматном порядке

Исполнителю самому надлежит выбрать, будут ли начинаться индексы в матрице с 0 или с 1.

6. Пример решения задачи

6.1. Индивидуальное задание:

- матрица содержит нули ниже главной диагонали;
- индексация начинается с 0.

6.2. Описание методов решения

6.2.1. Представление в памяти

Экономное использование памяти предусматривает, что для тех элементов матрицы, в которых наверняка содержатся нули, память выделяться не будет. Поскольку при этом нарушается двумерная структура матрицы, она может быть представлена в памяти как одномерный массив, но при обращении к элементам матрицы пользователь имеет возможность обращаться к элементу по двум индексам.

6.2.2. Модульная структура программного изделия

Программное изделие должно быть отдельным модулем, файл LAB2.C, в котором должны размещаться как данные (матрица и вспомогательная информация), так и функции, которые обеспечивают доступ. Внешний доступ к программам и данным модуля возможен только через вызов функций чтения и записи элементов матрицы. Доступные извне элементы программного модуля должны быть описаны в отдельном файле LAB2.H, который может включаться в программу пользователя оператором препроцессора:

```
#include "lab2.h"
```

Пользователю должен поставляться результат компиляции - файл LAB2.OBJ и файл LAB2.H. **6.2.3.** Преобразование 2-компонентного адреса элемента матрицы, которую задает пользователь, в 1-компонентную должно выполняться отдельной функцией (так называемой, функцией линеаризации), вызов которой возможен только из функций модуля. Возможны три метода преобразования адреса:

- при создании матрицы для нее создается также и дескриптор D[N] - отдельный массив, каждый элемент которого соответствует одной строке матрицы; дескриптор заполняется значениями, подобранными так, чтобы: n = D[x] + y, где x, y - координаты пользователя (строка, столбец), n - линейная координата;
- линейная координата подсчитывается методом итерации як сумма полезных длин всех строк, предшествующих строке x, и к ней прибавляется смещение y-го полезного элемента относительно начала строки;
- для преобразования подбирается единое арифметическое выражение, которой реализует функцию: n = f(x,y).

Первый вариант обеспечивает быстрейший доступ к элементу матрицы, ибо требует наименьших расчетов при каждом доступе, но плата за это - дополнительные затраты памяти на дескриптор. Второй вариант - наихудший по всем показателям, ибо каждый доступ требует выполнения оператора цикла, а это и медленно, и занимает память. Третий вариант может быть компромиссом, он не требует дополнительной памяти и работает быстрее, чем второй. Но выражение для линеаризации тут будет сложнее, чем первом варианте, следовательно, и вычисляться будет медленнее.

В программном примере, который мы приводим ниже, полностью реализован именно третий вариант, но далее мы показываем и существенные фрагменты программного кода для реализации и двух других.

6.3. Описание логической структуры

6.3.1. Общие переменные

В файле LAB2.C описаны такие статические переменные:

- int NN - размерность матрицы;
- int SIZE - количество ненулевых элементов в матрице;
- int *m_addr - адрес сжатой матрицы в памяти, начальное значение этой переменной - NULL - признак того, что память не выделена;
- int L2_RESULT - общий флаг ошибки, если после выполнения любой функции он равен -1, то произошла ошибка.

Переменные SIZE и m_addr описаны вне функций с квалификатором static, это означает, что вони доступны для всех функций в этом модуле, но недоступны для внешних модулей. Переменная L2_RESULT также описана вне всех функций, не без явного квалификатора. Эта переменная доступна не только для этого модуля, но и для всех внешних модулей, если она в них буде описана с квалификатором extern. Такое описание имеется в файле LAB2.H.

6.3.2. Функция creat_matr

Функция creat_matr предназначена для выделения в динамической памяти места для размещения сжатой матрицы. Прототип функции:

int creat_matr (int N);
где N - размерность матрицы.

Функция сохраняет значение параметра в собственной статической переменной и подсчитывает необходимый размер памяти для размещения ненулевых элементов матрицы. Для выделения памяти используется библиотечная функция C malloc. Функция возвращает -1, если при выделении произошла ошибка, или 0, если выделение прошло нормально. При этом переменной L2_RESULT также присваивается значение 0 или -1.

6.3.3. Функция close_matr

Функция close_matr предназначена для освобождения памяти при завершении работы с матрицей, Прототип функции:

int close_matr (void);

Функция возвращает 0 при успешном освобождении, -1 - при попытке освободить невыделенную память.

Если адрес матрицы в памяти имеет значения NULL, это признак того, что память не выделялась, тогда функция возвращает -1, иначе - освобождает память при помощи библиотечной функцииfree и записывает адрес матрицы - NULL. Соответственно функция также устанавливает глобальный признак ошибки - L2_RESULT.

6.3.4. Функция read_matr

Функция read_matr предназначена для чтения элемента матрицы. Прототип функции:

int read_matr(int x, int y);
где x и y - координаты (строка и столбец). Функция возвращает значение соответствующего элемента матрицы. Если после выполнения функции значение переменной L2_RESULT -1, то это указывает на ошибку при обращении.

Проверка корректности задания координат выполняется обращением к функции ch_coord, если эта последняя возвращает ненулевое значение, выполнение read_matr на этом и заканчивается. Если же координаты заданы верно, то проверяется попадание заданного элемента в нулевой или ненулевой участок. Элемент находится в нулевом участке, если для него номер строки больше, чем номер столбца. Если элемент в нулевом участке, функция просто возвращает 0, иначе - вызывает функцию линеаризации lin и использует значение, которое возвращает lin, как индекс в массиве m_addr, по которому и выбирает то значения, которое возвращается.

6.3.5. Функция write_matr

Функция write_matr предназначена для записи элемента в матрицу. Прототип функции:

int write_matr(int x, int y, int value);
где x и y - координаты (строка и столбец), value - то значение, которое нужно записать. Функция возвращает значение параметра value, или 0 - если была попытка записи в нулевой участок. Если после выполнения функции значение переменной L2_RESULT -1, то это указывает на ошибку при обращении.

Выполнение функции подобно функции read_matr с тем отличием, что, если координаты указывают на ненулевой участок, то функция записывает value в массив m_addr.

6.3.6. Функция ch_coord

Функция ch_coord предназначена для проверки корректности задания координат. Эта функция описана как static и поэтому может вызываться только из этого же модуля. Прототип функции:

```
static char ch_coord(int x, int y);
```

где x и y - координаты (строка и столбец). Функция возвращает 0, если координаты верные, -1 - если неверные. Соответственно, функция также устанавливает значение глобальной переменной L2_RESULT.

Выполнение функции собственно состоит из проверки трех условий:

- адрес матрицы не должен быть NULL, т.е., матрица должна уже находиться в памяти;
- ни одна из координат не может быть меньше 0;
- ни одна из координат не может быть больше NN.

Если хотя бы одно из этих условий не выполняется, функция устанавливает признак ошибки.

6.3.7. Функция lin

Функция lin предназначена для преобразования двумерных координат в индекс в одномерном массиве. Эта функция описана как static и поэтому может вызываться только из этого же модуля. Прототип функции:

```
static int lin(int x, int y);
```

где x и y - координаты (строка и столбец). Функция возвращает координату в массиве m_addr.

Выражение, значение которого вычисляет и возвращает функция, подобрано вот из каких соображений. Пусть мы имеет такую матрицу, как показано ниже, и нам нужно найти линейную координату элемента, обозначенного буквой A с координатами (x,y):

```
x x x x x x
0 x x x x x
0 0 x x A x
0 0 0 x x x
0 0 0 0 x x
0 0 0 0 0 x
```

Координату элемента можно определить как:

```
n = SIZE - sizeX +offY,
```

где SIZE - общее количество элементов в матрице (см. creat_matr),

```
SIZE = NN * (NN - 1) / 2 + NN;
```

sizeX - количество ненулевых элементов, которые содержатся в строке x и ниже,

```
sizeX = (NN - x) * (NN - x - 1) / 2 + (NN - x);
```

offY - смещение нужного элемента от начала строки x,

```
offY = y - x.
```

6.4. Программа пользователя

Для проверки функционирования нашего модуля создается программный модуль, который имитирует программу пользователя. Этот модуль обращается к функции creat_matr для создания матрицы нужного размера, заполняет ненулевую ее часть последовательно увеличивающимися числами, используя для этого функцию write_matr, и выводит матрицу на экран, используя для выборки ее элементов функцию read_matr. Далее в диалоговом режиме программа вводит запрос на свои действия и читает/пишет элементы матрицы с заданными координатами, обращаясь к функциям read_matr/write_matr. Если пользователь захотел закончить работу, программа вызывает функцию close_matr.

6.5. Тексты программных модулей

```
/********************* Файл LAB2.H   ************************/
/*     Описание функций и внешних переменных файла LAB2.C     */
extern int L2_RESULT;   /* Глобальна переменна - флаг ошибки */
/***** Выделение памяти под матрицу */
int creat_matr ( int N );
/***** Чтение элемента матрицы по заданным координатам */
int read_matr ( int x, int y );
/***** Запись элемент в матрицу по заданным координатам */
int write_matr ( int x, int y, int value );
/***** Уничтожение матрицы */
int close_matr ( void );
/****************** Конец файла LAB2.H   ************************/

/************************ Файл LAB2.C ************************/
/* В этом файле определены функции и переменные для обработки
   матрицы, заполненной нулями ниже главной диагонали          */
#include <alloc.h>
static int NN;                      /* Размерность матрицы */
static int SIZE;                        /* Размер памяти */
static int *m_addr=NULL;              /* Адрес сжатой матрицы */
static int lin(int, int);     /* Описание функции линеаризации */
static char ch_coord(int, int);   /* Описание функции проверки */
int L2_RESULT;              /* Внешняя переменная, флаг ошибки */

/*********************************************************/
/*           Выделение памяти под сжатую матрицу        */
int creat_matr ( int N ) {
  /* N - размер матрицы */
  NN=N;
  SIZE=N*(N-1)/2+N;
  if ((m_addr=(int *)malloc(SIZE*sizeof(int))) == NULL )
    return L2_RESULT=-1;
  else
    return L2_RESULT=0;
/* Возвращает 0, если выделение прошло успешно, иначе -1 */
}
/*********************************************************/
/*       Уничтожение матрицы (освобождение памяти)          */
int close_matr(void) {
  if ( m_addr!=NULL ) {
```

```
      free(m_addr);
      m_addr=NULL;
      return L2_RESULT=0;
      }
   else return L2_RESULT=-1;
/*  Возвращает 0, если освобождение пршло успешно, иначе - -1  */
}
/*************************************************************/
/*     Чтение элемента матрицы по заданным координатам     */
int read_matr(int x, int y) {
   /* x, y -координати (строка, столбец) */
   if ( ch_coord(x,y) ) return 0;
   /* Если координаты попадают в нулевой участок - возвращается
      0, иначе - применяется функция линеаризации */
   return (x > y) ? 0 : m_addr[lin(x,y)];
   /* Проверка успешности чтения - по переменной
      L2_RESULT:  0 - без ошибок, -1 - была ошибка */
}

/*************************************************************/
/*      Запись элемента матрицы по заданным координатам      */
int write_matr(int x, int y, int value) {
   /* x, y -координати, value - записываемое значение */
   if ( chcoord(x,y) ) return;
   /* Если координаты попадают в нулевой участок - записи нет,
      иначе - применяется функция линеаризации */
   if ( x > y ) return 0;
   else return m_addr[lin(x,y)]=value;
   /* Проверка успешности записи - по L2_RESULT */
}

/*************************************************************/
/*       Преобразование 2-мерних координат в линейную       */
/*                    (вариант 3)                            */
static int lin(int x, int y) {
   int n;
    n=NN-x;
   return SIZE-n*(n-1)/2-n+y-x;
}

/*************************************************************/
/*                Проверка корректности обращения            */
static char ch_coord(int x, int y) {
   if ( ( m_addr==NULL ) ||
        ( x>SIZE ) || ( y>SIZE ) || ( x<0 ) || ( y<0 ) )
      /* Если матрица не размещена в памяти, или заданные
         координаты выходят за пределы матрицы */
       return L2_RESULT=-1;
    return L2_RESULT=0;
}
/**********************Конец файла LAB2.C **********************/

/************************ Файл MAIN2.C ************************/
```

```
/* "Программа пользователя" */
#include "lab2.h"
main(){
 int R;    /* размерность */
 int i, j; /* номера строки и столбца */
 int m;    /* значения элемента */
 int op;   /* операция */
  clrscr();
  printf('Введите размерность матрицы >'); scanf("%d",R);
  /* создание матрицы */
  if ( creat_matr (R) ) {
     printf("Ошибка создания матрицы\n");
     exit(0);
     }
  /* заполнение матрицы */
  for ( m=j=0; j<R; j++)
     for ( i=o; i<R; i++)
          write_matr(i,j,++m);
   while(1) {
      /* вывод матрицы на экран */
      clrscr();
      for (j=0; j<R; j++) {
         for (i=0; i<R; i++)
            printf("%3d ",read_matr(i,j));
            printf("\n");
         }
      printf("0 - выход\n1 - чтение\n2 - запись\n>")
      scanf("%d",&op);
      switch(op) {
        case 0:
         if (close_matr()) printf("Ошибка при уничтожении\n");
         else printf("Матрица уничтожена\n");
         exit(0);
        case 1: case 2:
         printf("Введите номер строки >");
         scanf("%d",&j);
         printf("Введите номер столбца >");
         scanf("%d",&i);
         if (op==2) {
            printf("Введите значение элемента >");
            scanf("%d",&m);
            write_matr(j,i,m);
            if (L2_RESULT<0) pritnf("Ошибка записи\n");
            }
         else {
            m=read_matr(j,i);
            if (L2_RESULT<0) pritnf("Ошибка считывания\n");
            else printf("Считано: %d\n",m);
            }
         printf("Нажмите клавишу\n"); getch();
         break;
        }
      }
```

```
}
/*********************Конец файла MAIN2.C **********************/
```

6.6. Варианты.

Ниже приведены фрагменты программных кодов, которые отличают варианты, рассмотренные в 6.2.3.

Вариант 1 требует:

- добавления к общим статическим переменным еще переменной:
- `static int *D;  /* адрес дескриптора */`
- добавления такого блока в функцию creat_matr:
- `{`
- `int i, s;`
- `D=(int *)malloc(N*sizeof(int));`
- `for (D[0]=0,s=NN-1,i=1; i<NN; i++)`
- `D[i]=D[i-1]+s--;`
- `}`
- изменения функции lin на:
- `static int lin(int x, int y) {`
- `return D[x]+y;`
- `}`

Вариант 2 требует:

- изменения функции lin на:
- `static int lin(int x, int y) {`
- `int s;`
-
- `for (s=j=0; j<x; j++)`
- `s+=NN-j;`
- `return s+y-x;`

`}`

ЛАБОРАТОРНАЯ РАБОТА №13 - 14

ПРОВЕРКА ОБОРУДОВАНИЯ

1. Цель работы

Получение практических навыков в определении конфигурации и основных характеристик ПЭВМ.

2. Темы для предварительного изучения

- Конфигурация ПЭВМ.
- Склад, назначение и характеристики основных модулей ПЭВМ.

3. Постановка задачи

Для компьютера на своем рабочем месте определить:

- тип компьютера;
- конфигурацию оборудования;
- объем оперативной памяти;
- наличие и объем расширенной памяти;
- наличие дополнительных ПЗУ;
- версию операционной системы.

4. Порядок выполнения

Порядок выполнения работы и содержание отчета определены в общих указаниях.

5. Пример решения задачи

5.1. Структура данных программы

Программа использует, так называемый, список оборудования - 2-байтное слово в области данных BIOS по адресу 0040:0010. Назначение разрядов списка оборудования такое:

Биты	Содержимое
0	установлен в 1, если есть НГМД (см.разряды 6, 7)
1	установлен в 1, если есть сопроцессор
2,3	число 16-Кбайтних блоков ОЗУ на системной плате
4,5	код видеоадаптера: 11 - MDA, 10 - CGA, 80 колонок, 01 - CGA, 40 колонок, 00 - другой
6,7	число НГМД-1 (если в разряде 0 единица)
8	0, если есть канал ПДП
9,10,11	число последовательных портов RS-232
12	1, если есть джойстик
13	1, если есть последовательный принтер
14,15	число параллельных принтеров

5.2. Структура программы

Программа состоит только из основной функции main(). Выделения фрагментов программы в отдельные процедуры не требуется, потому что нет таких операций, которые во время работы программы выполняются многократно.

5.3. Описание переменных

Переменные, применяемые в программе:

- type_PC - байт типа компьютера, записанный в ПЗУ BIOS по адресу FF00:0FFE;
- a, b - переменные для определения объема extended-памяти ПЭВМ, a - младший байт, b - старший байт;
- konf_b - 2-байтное слово из области данных BIOS, которое содержит список оборудования;
- type - массив символьных строк, представляющих типы компьютера;
- typ1A - массив байт, содержащий коды типов дисплеев;
- types1A[] - массив строк, содержащий названия типов дисплеев;
- j - вспомогательная переменная, которая используется для идентификации типа дисплея;
- seg - сегмент, в котором размещено дополнительное ПЗУ;
- mark - маркер ПЗУ;
- bufVGA[64] - буфер данных VGA, из которого (при наличии VGA) ми выбираем объем видеопамяти;
- rr и sr - переменные, которые используются для задания значения регистров общего назначения и сегментных регистров, соответственно, при вызове прерывания.

5.4. Описание алгоритма программы

Алгоритм основной программы может быть разбито на 5 частей.

Часть 1 предназначена для определения типа компьютера. Для этого прочитаем байт, записанный в ПЗУ BIOS по адресу FF00:0FFE. В зависимости от значения этого байта сделаем вывод о типе ПЭВМ. Так, например, компьютеру типа AT соответствует код 0xFC.

Часть 2 предназначена для определения конфигурации ПЭВМ. Для этого прочитаем из области данных BIOS список оборудования. Для определения количества дисководов (если бит 0 установлен в 1) необходимо выделить биты 6 и 7 (маска 00C0h) и сместить их вправо на 6 разрядов, а потом добавить 1.

Для определения количества 16-Кбайтних блоков ОЗУ на системной плате необходимо выделить биты 2 и 3 с помощью маски 000Ch, сместить вправо на 2 разряды и добавить 1.

Для определения количества последовательных портов RS-232 выделить с помощью маски 0Eh биты 9-11 и сместить вправо на 9 разрядов.

Для определения наличия математического сопроцессора - проверить установку бита 1 маской 0002h.

Для определения наличия джойстика - бита 12 с помощью маски 1000h.

Определить количество параллельных принтеров можно, выделив биты 14 и 15 маской C000h и сместив их вправо на 14 разрядов.

Поскольку список оборудования содержит недостаточно информации про дисплейный адаптер, то для уточнения типа адаптера выполним дополнительные действия.

Видеоадаптер обслуживается прерыванием BIOS 10h. Для новых типов адаптеров список его функций расширяется. Эти новые функции и используются для определения типу адаптера.

Функция 1Ah доступна только при наличии расширения BIOS, ориентированного на обслуживание VGA. В этом случае функция возвращает в регистре AL код 1Ah - свою "визитную карточку", а в BL - код активного видеоадаптера. В случае, если функция 1Ah поддерживается, обратимся еще к функции 1Bh - последняя заполняет 70-байтний блок информации про состояние, из которого мы выбираемо объем видеопамяти.

Если 1Ah не поддерживается, это означает, что VGA у нас нет, в этом случае можно обратиться к функции 12h - получение информации про EGA. При наличии расширения, ориентированного на EGA, эта функция изменяет содержимое BL (перед обращением он должен быть 10h) на 0 (цветной режим) или на 1 (монохромный режим) а в BH возвращает объем видеопамяти.

Если же ни 1Ah, ни 12 не поддерживаются, то список оборудования BIOS содержит достаточную информацию про видеоадаптер и, выделивши биты 4, 5 мы можем сделать окончательный вывод про тип адаптера, который у нас есть.

В третьей части программы определим объем оперативной памяти, наличие и объем extended-памяти. Объем оперативной памяти для AT может быть прочитан из регистров 15h (младший байт) и 16h (старший байт) CMOS-памяти или из области памяти BIOS по адресу 0040:0013 (2-байтное слово). Кроме того, в ПЭВМ может быть еще и дополнительная (expanded) память свыше 1 Мбайту. Ее объем можно получит из регистров 17h (младший байт) и 18h (старший байт) CMOS-памяти. Для чтения регистра CMOS-памяти необходимо видать в порт 70h байт номера регистра, а потом из порта 71h прочитать байт содержимого этого регистра.

В следующей части программы определим наличие и объем дополнительных ПЗУ. В адресному пространстве от C000:0000 по F600:0000 размещаются расширения ПЗУ (эта память не обязательно присутствует в ПЭВМ). Для определения наличия дополнительного ПЗУ будем читать первое слово из каждых 2 Кбайт, начиная с адреса C000:0000 в поисках маркера расширения ПЗУ: 55AAh. Если такой маркер найден, то следующий байт содержит длину модуля ПЗУ.

В заключительной части программы определим версию DOS, установленную на ПЭВМ. Для этого воспользуемся функцией DOS 30h, которая возвращает в регистре AL старшее число номера версии, а в регистре AH - младшее число.

5.5. Текст программы

```
/*----------------Лабораторная работа N4------------------*/
/*----------"Проверка состава оборудования"--------------*/

/* Подключение стандартных заголовков */
#include <dos.h>
#include <conio.h>
#include <stdio.h>

/*-------------------------------------------------------*/
```

```
void main()
{
 unsigned char type_PC, /* Тип компьютера              */
               a,b;     /* Переменные для определения  */
                        /* характеристик памяти ПЭВМ   */
 unsigned int  konf_b;  /* Байт конфигурации из BIOS   */
 char *type[]={"AT","PCjr","XT","IBM PC","unknown"};
 unsigned char typ1A[]={0,1,2,4,5,6,7,8,10,11,12,0xff};
 char *types1A[]={"нема дисплею","MDA, моно","CGA, цв.",
                  "EGA, цв.","EGA, моно","PGA, цв.",
                  "VGA, моно, анал.","VGA, кол., анал.",
                  "MCGA, кол., цифр.","MCGA, моно, анал."
                  "MCGA, кол., анал.","неизвестный тип",
                  "непредусмотренный код"};
 unsigned int j;       /* Вспомогательная переменная */
 unsigned int seg;     /* Сегмент ПЗУ                */
 unsigned int mark=0xAA55;   /* Маркер ПЗУ           */
 unsigned char bufVGA[64];       /* Буфер данных VGA */
 union REGS rr;
 struct SREGS sr;

 textbackground(0);
 clrscr();
 textattr(0x0a);
 cprintf("Лабораторная работа N5");
 cprintf("\nПроверка состава оборудования");

 /* Определения типа компьютера */
 type_PC=peekb(0xF000,0xFFFE);
 if( (type_PC-=0xFC)>4)
       type_PC=4;
 textattr(0x0b);
 cprintf("\nТип компьютера: ");
 textattr(0x0f);
 cprintf("%s\n\r",type[type_PC]);

 /* Конфигурация*/
 konf_b=peek(0x40,0x10); /* Чтение байта оборудования */
                         /* из памяти BIOS            */
 textattr(0x0b);
 cprintf("Конфигурация:\n\r");

 /* Количество дисководов */

 textattr(0x0e);
 cprintf("   Дисководов ГМД:           ");
 textattr(0x0f);
 if(konf_b&0x0001)
       cprintf("%d\n\r",((konf_b&0x00C0)>>6)+1);
 else
   cprintf("нет\n\r");
 textattr(0x0e);
 cprintf("   Математич. сопроцессор:   ");
```

```
textattr(0x0f);
if(konf_b&0x0002)
  cprintf("есть\n\r");
else
  cprintf("нет\n\r");
textattr(0x0e);
cprintf("   Тип дисплейного адаптера: ");
textattr(0x0f);

/* Определение активного адаптера */
/* Предположим наличие VGA */
rr.h.ah=0x1a;
rr.h.al=0;
int86(0x10,&rr,&rr);
if(rr.h.al==0x1a)  /* Поддерживается функция 1Ah */
{                  /* прерывания 10h          */
 for(j=0;j<12;j++)
  if(rr.h.bl==typ1A[j])
   break;
 cprintf("%s",types1A[j]);

 if(j>0 && j<12)
 {
  rr.h.ah=0x1b;
  rr.x.bx=0;
  sr.es=FP_SEG(bufVGA);
  rr.x.di=FP_OFF(bufVGA);
  int86x(0x10,&rr,&rr,&sr);
  cprintf(", %d Кбайт\n\r",((int)bufVGA[49]+1)*64);
 }
 else
       cprintf("\n\r");
}
else
{
 /* Предположим наличие EGA */
 rr.h.ah=0x12;
 rr.h.bl=0x10;
 int86(0x10,&rr,&rr);
 if(rr.h.bl!=0x10) /* Поддерживается функция 12h */
 {                 /* прерывания 10h */
  cprintf("EGA");
  if(rr.h.bh)
   cprintf(" моно");
  else
   cprintf(" кол.");
  cprintf(", %d Кбайт\n\r",((int)rr.h.bl+1)*64);
 }
 else
 {
  /* CGA или MDA */
  switch(konf_b&0x0030)
  {
```

```
   case 0:    cprintf("EGA/VGA\n\r");break;
   case 0x10: cprintf("CGA,40\n\r");break;
   case 0x20: cprintf("CGA,80\n\r");break;
   case 0x30: cprintf("MDA");break;
  }
 }
}


/* Блоки ОЗУ на системной плате */
textattr(0x0e);
cprintf("\n\r   Первичный блок памяти:    ");
textattr(0x0f);
switch (konf_b&0x000C)
{
 case 0:cprintf("16 Кбайт\n\r");break;
 case 4:cprintf("32 Кбайт\n\r");break;
 case 8:cprintf("48 Кбайт\n\r");break;
 case 12:cprintf("64 Кбайт или больше\n\r");break;
}

/* Количество последовательных портов RS-232 */
textattr(0x0e);

cprintf("   Портов RS232:             ");
textattr(0x0f);
cprintf("%d\n\r",(konf_b&0x0E00)>>9);

/* Наличие джойстика */
textattr(0x0e);
cprintf("   Джойстик:                 ");
textattr(0x0f);
if(konf_b&0x1000 )
 cprintf("есть\n\r");
else
 cprintf("нет\n\r");

/* Количество параллельних принтеров */
textattr(0x0e);
cprintf("   Принтеров:                ");
textattr(0x0f);
cprintf("%d\n\n\r",(konf_b&0xC000)>>14);

/* Объем оперативной памяти */

textattr(0x0e);
cprintf("Объем оперативной памяти: ");
textattr(0x0f);
cprintf("%d Кбайт\n\r",peek(0x40,0x13));
textattr(0x0e);

/* Наличие и объем extended-памяти */
outportb(0x70,0x17);
```

```
 a=inport(0x71);
 outportb(0x70,0x18);
 b=inport(0x71);
 cprintf("Объем extended-памяти:    ");
 textattr(0x0f);
 cprintf("%d  Кбайт\n\n\r",(b<<8)|a);


 /* Наличие дополнительных ПЗУ */
 for( seg=0xC000;seg<0xFFB0;seg+=0x40)
 /* Просмотр памяти от C000:0 с шагом 2 К */
  if(peek(seg,0)==mark)   /* Маркер найден */
  {
   textattr(0x0a);
   cprintf("Адрес ПЗУ =");
   textattr(0x0f);
   cprintf(" %04x",seg);
   textattr(0x0a);
   cprintf(". Длина модуля = ");
   textattr(0x0f);
   cprintf("%d",512*peekb(seg,2));
   textattr(0x0a);

   cprintf(" байт\n\r",peekb(seg,2));
  }

 /* Определение версии операцийной системы */
 rr.h.ah=0x30;

 intdos(&rr,&rr);
 textattr(0x0c);
 cprintf("\n\rВерсия MS-DOS ");
 textattr(0x0f);
 cprintf("%d.%d\n\r",rr.h.al,rr.h.ah);

 textattr(0x0a);
 gotoxy(30,24);
 cprintf("Нажмите любую клавишу");
 textattr(0x07);
 getch();
 clrscr();
}
```

5.6. Результаты работы программы

В процессе работы программы на экран была выведена такая информация:

```
                    Лабораторная работа N4
                  Проверка состава оборудования

Тип компьютера: AT
Конфигурация:
  Дисководов ГМД:          2
```

Математич. сопроцессор: есть
Тип дисплейного адаптера: VGA, кол., анал., 256 Кбайт

Первичный блок памяти: 16 Кбайт
Портов RS232: 2
Джойстик: нет
Принтеров: 1

Объем оперативной памяти: 639 Кбайт
Объем extended-памяти: 384 Кбайт

Адрес ПЗУ = c000. Длина модуля = 24576 байт

Версия MS-DOS 6.20

ЛАБОРАТОРНАЯ РАБОТА №15 - 16

УПРАВЛЕНИЕ ПАМЯТЬЮ

1. Цель работы

Просмотр таблицы векторов прерываний, которая в данный момент находится в ПЭВМ на рабочем месте.

2. Темы для предварительной проработки

- Таблица векторов прерываний.
- Управление памятью.

3. Постановка задачи

Определить, какие вектора прерываний на данном компьютере перехвачены программами не из состава DOS/BIOS и имена этих программ.

4. Порядок выполнения работы

5. Пример решения задачи

5.1. Разработка алгоритма решения

5.1.1. Структура программы

Программа состоит из основной программы main() и пяти вспомогательных функций.

- void *readvect(int in) - функция читает вектор прерывания с номером in и возвращает его значение.
- void get_memtop(void) - функция возвращает адрес начала цепочки MCB csegm.
- void dos_version_h(void) - функция возвращает номер версии DOS.
- void name_handler(void) - функция находит какой программой было перехвачено текущее прерывание.
- void PrintVec(int num) - функция выводит на экран номера прерываний и программы которые их перехватили начиная с номера num.

5.1.2. Описание переменных

Переменные глобальные для всей программы:

- memtop - сегментный адрес начала памяти;
- csegm - сегментный адрес текущего MCB;
- othersegm - сегментный адрес другого MCB; 8
- fathersegm - сегментный адрес родител0я;
- *envstr - адрес строки окружения;
- envlen - длина очередной строки окружения;
- envsize - размер блока окружения;
- dos - номер версии DOS;
- *vect - вектор;
- intnum - номер прерывания.

Структура MCB:

```
struct MCB {
  byte type;        /* тип */
  word owner;       /* владелец */
  word size;        /* размер */
  byte reserved[3]; /* не используется */
  char pgmname[8];  /* имя (только DOS 4.0 и выше) */
  };
```

5.1.3. Описание алгоритма программы

Главная программа main() осуществляет бесконечный цикл для обработки кодов нажатых клавиш и в зависимости от них вызывается функция PrintVec(int num) для вывода на экран информации о векторах прерывания. Выход из данного цикла производится по нажатию клавиши Esc.

Функция readvect() читает вектор заданного прерывания. Для чтения вектора используем функцию 35h DOS (прерывание 21h):

Вход:	AH = 35h; AL = номер вектора прерывания.
Выход:	ES:BX = адрес программы обработки прерывания.

Функция get_memtop() возвращает адрес начала цепочки MCB csegm. Адрес начала цепочки блоков памяти можно получить при помощи недокументированной функции DOS 52h. Эта функция возвращает в регистрах ES:BX некоторый адрес. Вычтя из этого адреса 2, получим адрес того слова памяти, в котором DOS хранит сегментный адрес первого MCB в цепочке.

Функция dos_version_h() возвращает номер версии DOS при помощи функции 30h.

Функция name_handler() находит какой программой было перехвачено текущее прерывание.

Функция PrintVec(int num) выводит на экран номера прерываний и программы которые их перехватили начиная с номера num, при этом она в цикле вызывает функцию name_handler() для получения информации о программе, которая перехватила текущий вектор прерывания.

5.2 Текст программы

```
/*-----------------Лабораторная работа N13----------------*/
/*-----------------"Управление памятью."-----------------*/

/* Подключение стандартных заголовков */
#define byte unsigned char
#define word unsigned int
#define Esc 27
#define Up 0x48
#define Down 0x50
#define Home 0x47
#define Endk 79
#define PgUp 73
```

```
#define PgDn 81

#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>

void *readvect(int in);
void get_memtop(void);
void dos_version_h(void);
void name_handler(void);
void PrintVec(int num);

struct MCB { /* блок управления памятью */
  char type; /* тип */
  word owner,/* владелец */
       size; /* размер */
  byte reserved[3]; /* зарезервировано */
  char pgmname[8];  /* имя программы, находящейся в данном блоке
                       ( для  DOS 4.0 и выше )         */
   };

struct MCB *cmcb;  /* адрес текущего MCB */
struct MCB *emcb;  /* адрес MCB среды */
word memtop;       /* сегментныйадрес начала памяти */
word csegm;        /* сегментный адрес текущего MCB */
word othersegm;    /* сегментныйадрес другого MCB */
word othersegm1;
word fathersegm;   /* сегментныйадрес родителя */
byte *envstr;      /* адрес строки окружения */
int envlen;        /* длина очередной строки окружения */
int envsize;       /* размер блока окружения */
byte dos;          /* номер версии DOS */
void far *vect;    /* вектор */
int intnum;      /* номер прерывания */

union REGS rr;
struct SREGS sr;

void main() {
 int i,a;

get_memtop();
dos_version_h();
clrscr();
textattr(10);
clrscr();
cprintf("---------------");
cprintf("             Лабораторная работа N13             ");
cprintf("---------------");
cprintf("---------------");
cprintf("                Управление памятью.              ");
cprintf("---------------");
```

```
gotoxy(28,3);
textattr(12);
cprintf("Таблица векторов прерывания.");
gotoxy(1,4);
textattr(11);
cprintf("Номер   - Адрес     - Чем_занят");
gotoxy(7,25);
textattr(7);
cprintf("Клавиши управления: <Up>,<Dn>,<Home>,<End>,<PgUp>,<PgDn>
        ,<Esc>");

  intnum=0;
  PrintVec(intnum);
  for(i=0;i==0;)
  switch(getch())
  { /* Обработка нажатых клавиш */
   case Esc: i++; break;
   case 0: switch (getch()) {
                          case   Up:intnum--;
                  if(intnum<0) intnum=0;
                  PrintVec(intnum);
                  break;
           case Down:intnum++;
                  if(intnum>240) intnum=240;
                  PrintVec(intnum);
                  break;
           case PgUp:intnum-=16;
                  if(intnum<0) intnum+=16;
                  PrintVec(intnum);
                  break;
           case PgDn:intnum+=16;
                  if(intnum>256) intnum-=16;
                  PrintVec(intnum);
                  break;
           case Home:intnum=0;
                  PrintVec(intnum);
                  break;
           case Endk:intnum=0xf0;
                  PrintVec(intnum);
                  break;
         }
  }
}
/*--------------------------------------------------------*/
void PrintVec(int num)
{
int i,y;
  gotoxy(1,5);
  textattr(14);
  for (i=0;i<17;i++)
  { y=wherey();
    cprintf("                                    \n");
```

```
   gotoxy(1,y);
   if ((num+i)<=0xff)
    {
     vect=readvect(num+i);
     cprintf("INT %02Xh - %Fp - ",num+i,vect);
     if (vect==NULL)
      cprintf("%-16s","Свободен");
     else
     {intnum=num;
      name_handler();}
      printf("\n");
     }
    }
}
/*-------------------------------------------------------*/
void name_handler(void)
{
int i;
char *s;
word tsegm;

 othersegm1=FP_SEG(vect);
 tsegm=memtop;
 do
 {
  csegm=tsegm;
  cmcb=(struct MCB *)MK_FP(csegm,0);
  tsegm=csegm+cmcb->size+1;
 }
 while((othersegm1>tsegm)&&(cmcb->type!='Z'));
 if ((cmcb->owner<memtop)||(cmcb->type=='Z'))
  printf("%-16s","Занят Bios/Dos");
 else
   {  /* блок не принадлежит DOS */
     othersegm=peek(cmcb->owner,0x2c);
     fathersegm=peek(cmcb->owner,0x16);
     /* если хозяин сам себе родитель, то это COMMAND */
     if (cmcb->owner==fathersegm)
      printf("%-16s","COMMAND.COM");
     else
     { /* для другой программы узнаем ее имя */
      if (dos>3)
       {
        emcb=(struct MCB *)MK_FP(cmcb->owner-1,0);
        for (i=0,s=emcb->pgmname; i<8; i++)
        {
          if (*s>0)
           printf("%c",*(s++));
          else
           printf(" ");
        }
        printf("%-8s"," ");
       }
```

```
        else
          if (dos>2)
          { /* для DOS 3.0 и выше имя - из строки вызова */
           emcb=(struct MCB *)MK_FP(othersegm-1,0);
           envsize=emcb->size*16; /*размер окружения */
           envstr=(char *)MK_FP(othersegm,0);
           do
            {
             /* пропуск строк окружения до пустой строки */
             envlen=strlen(envstr)+1;
             envstr+=envlen; envsize-=envlen;
            }
           while ((envlen>1)&&(envsize>0));
           envstr+=2;
           envsize-=2; /* 2 байта - кол.строк */
           /* envstr - указатель на строку вызова */
           if (envsize>0)
            printf("%-16s",envstr);
          }
      }
     }
}

/*==== Получение вектора ====*/
void *readvect(int in) {
 union REGS rr; struct SREGS sr;
  rr.h.ah=0x35; rr.h.al=in; intdosx(&rr,&rr,&sr);
  return(MK_FP(sr.es,rr.x.bx));
}

/* получить адрес начала цепочки MCB csegm */
void get_memtop(void)
{
  rr.h.ah=0x52;
  intdosx(&rr,&rr,&sr);
  memtop=csegm=peek(sr.es,rr.x.bx-2);
}
/* получить номер версии DOS */
void dos_version_h(void)
{
  rr.h.ah=0x30;
  intdos(&rr,&rr);
  dos=rr.h.al;
}
```

5.3. Результаты работы программы

В процессе работы программы на экран была выведена таблица векторов прерываний, просматривать которую можно с помощью клавиш управления:

```
--------------------------------------------------------------------------------
---------------          Лабораторная работа N13          ---------------
---------------          Управление памятью.              ---------------
                      Таблица векторов прерывания.
Номер   - Адрес     - Чем_занят
INT 00h - 0BB0:0125 - TC_LAB13
INT 01h - 0070:06F4 - Занят Bios/Dos
INT 02h - 03DA:0016 - Занят Bios/Dos
INT 03h - 0070:06F4 - Занят Bios/Dos
INT 04h - 0070:06F4 - Занят Bios/Dos
INT 05h - F000:FF54 - Занят Bios/Dos
INT 06h - F000:EB43 - Занят Bios/Dos
INT 07h - F000:EAEB - Занят Bios/Dos
INT 08h - 0ACF:013D - COPY_ECR
INT 09h - 0ACF:0117 - COPY_ECR
INT 0Ah - 03DA:0057 - Занят Bios/Dos
INT 0Bh - 03DA:006F - Занят Bios/Dos
INT 0Ch - 03DA:0087 - Занят Bios/Dos
INT 0Dh - 03DA:009F - Занят Bios/Dos
INT 0Eh - 03DA:00B7 - Занят Bios/Dos
INT 0Fh - 0070:06F4 - Занят Bios/Dos
INT 10h - 057A:035B - KEYRUS

     Клавиши управления: <Up>,<Dn>,<Home>,<End>,<PgUp>,<PgDn>,<Esc>
--------------------------------------------------------------------------------
```