Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

**Е. В. Калабухов, И. В. Лукьянова,
А. Г. Третьяков**

КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ЯЗЫК ПРОГРАММИРОВАНИЯ АССЕМБЛЕР

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-40 02 01 «Вычислительные машины, системы и сети»

Минск БГУИР 2016

УДК 004.438:004.42(076.5)
ББК 32.973.26-018.1я73+32.973.26-018.2я73
К17

Р е ц е н з е н т ы:
кафедра информационных систем управления факультета прикладной математики и информатики Белорусского государственного университета (протокол №10 от 11.05.2015);

доцент кафедры информационных систем и технологий учреждения образования «Белорусский государственный технологический университет», кандидат технических наук, доцент Н. В. Пацей

Калабухов, Е. В.

К17 Конструирование программ и языки программирования. Язык программирования Ассемблер : пособие / Е. В. Калабухов, И. В. Лукьянова, А. Г. Третьяков. – Минск : БГУИР, 2016. − 80 с. : ил.
ISBN 978-985-543-188-7.

Предназначено для проведения занятий по дисциплине «Конструирование программ и языки программирования» с использованием языка программирования Ассемблер. В пособии приведены описания лабораторных работ с рассмотрением кратких теоретических сведений по изучаемому материалу и примеров программ, поясняющих их. Также присутствуют контрольные вопросы, дающие возможность определить подготовленность студента к выполнению задания. Приведены варианты заданий.

УДК 004.438:004.42(076.5)
ББК 32.973.26-018.1я73+32.973.26-018.2я73

ISBN 978-985-543-188-7

Введение

Учебная дисциплина «Конструирование программ и языки программирования» является одной из базовых в рамках обучения студентов методикам и способам создания программного обеспечения. Данная дисциплина закладывает фундаментальные основы организации и создания программного обеспечения, является связующим звеном между простыми методиками структурного программирования и узкоспециализированными направлениями в системах программирования.

Данное пособие предназначено для формирования навыков создания и отладки программ, написанных на языке ассемблера. Изучение языка ассемблера является необходимой частью подготовки профессиональных программистов, поскольку позволяет шире понять принципы работы ЭВМ, операционных систем и трансляторов с языков высокого уровня. В данном пособии изучается Ассемблер для процессора Intel 8086, являющегося базой для построения других, более современных, процессоров Intel широко распространенных в настоящий момент.

Пособие построено по принципу «от простого к сложному», поэтому выполнять лабораторные задания желательно последовательно. Для получения большей пользы также желательно соблюдать следующий общий порядок выполнения лабораторной работы:

1) получить задание на лабораторную работу;
2) изучить краткие теоретические сведения;
3) ознакомиться с материалами литературных источников;
4) ответить на контрольные вопросы;
5) разработать алгоритм программы;
6) написать и отладить программу;
7) выполнить функциональное тестирование программы;
8) оформить отчет и защитить работу.

Лабораторная работа №1

Тема работы: создание простой программы на языке ассемблера.

Цель работы: ознакомиться с программным обеспечением, предназначенным для сборки, отладки и запуска программ на языке ассемблера. Ознакомиться с основными особенностями архитектуры процессора и общей структурой программы.

Теоретические сведения

Ассемблер – это программа, которая переводит мнемонические текстовые команды в машинный код. Для формирования программ на языке ассемблера можно использовать разные среды разработки:

1. TASM (turbo assembler) – система программирования фирмы Borland, предназначенная для создания 16-битных программ для DOS и процессоров семейства Intel x86.

TASM включает в себя следующие основные компоненты:

- компилятор языка Ассемблер – *tasm.exe* – программа, предназначенная для компиляции файла, написанного на языке ассемблера (*.asm*) в объектные модули (*.obj*);
- компоновщик (linker) – *tlink.exe* – программа, предназначенная для сборки исполняемого файла из объектных модулей и библиотек;
- отладчик (debugger) – *td.exe* – программа, предназначенная для отладки созданных исполняемых файлов.

Все указанные выше программы получают опции через командную строку (все доступные функции можно получить, запустив программу без параметров). Кроме этого, Borland предоставляет интегрированную среду асемблера *ta.exe* – программу, объединяющую текстовый редактор и указанные выше компоненты и похожую по своим функциональным возможностям на среду «*Turbo C*».

2. MASM – система программирования от Microsoft, схожая с TASM.

3. FASM (flat assembler) – свободно распространяемый многопроходный ассемблер.

4. Эмулятор Emu8086 – система программирования, предназначенная для создания и отладки ассемблерных программ под Windows, имеет более дружественный интерфейс, чем DOS-среды, но работает достаточно медленно, а также имеет некоторые ограничения на работу с ресурсами.

5. Среда разработки программ для языка C\C++ (например Borland C++ или MS Visual Studio) – в таких средах можно использовать команды встроенного ассемблера, что позволит изучить некоторые аспекты программирования на ассемблере без создания полноценной программы полностью на ассемблере.

Также следует понимать, что создание программ на ассемблере должно в требуемой мере использовать ресурсы и возможности операционной системы

для взаимодействия с аппаратными средствами компьютера. Для лабораторных работ актуально использование операционной системы MS-DOS или FreeDOS – это будет актуально при работе с DOS-средами разработки ассемблерных программ, а также для создания резидентных программ. Для того чтобы не устанавливать отдельно DOS на компьютере с Windows (Unix), можно воспользоваться виртуальной машиной (например Oracle VM VirtualBox) или средой DOSBox.

Для создания простейших программ на языке ассемблера требуется знание следующих понятий:

1. *Регистры процессора.*

В состав процессора Intel 8086 входят 16-битовые регистры:

- общего назначения – **AX** (аккумулятор), **BX** (база), **CX** (счетчик), **DX** (данные), также можно обращаться отдельно к их старшей или младшей 8-битовой половине: **AH**, **AL**, **BH**, **BL**, **CH**, **CL**, **DH** и **DL**;

- сегментные – **CS** (код), **DS** (данные), **SS** (стек) и **ES** (расширенные данные);

- адресные – **SI** (индекс источника), **DI** (индекс приемника) и **BP** (указатель базы); также к ним относят и регистр **BX** (база);

- указатель стека – **SP**;

- указатель команд – **IP**;

- регистр флагов – **FLAGS** (биты признаков, рис. 1).

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

Рис. 1. Формат регистра флагов

Важные флаги:

- **CF** – флаг переноса;
- **ZF** – флаг нуля;
- **SF** – флаг знака;
- **OF** – флаг переполнения.

2. *Сегментная адресация памяти.*

Для адресации памяти процессор Intel 8086 использует 16-разрядные адресные регистры, что обеспечивает доступ к 64 Кбайт (0000_{16}–$FFFF_{16}$) основной памяти. Так как все возможное адресное пространство памяти данного процессора составляет 1 Мбайт (00000_{16}–$FFFFF_{16}$), то при этом память приходится делить на сегменты объемом 64 Кбайт. Каждый сегмент начинается на границе параграфа (16 байт) от начала памяти, поэтому стартовыми для сегментов могут быть следующие адреса памяти: 00000_{16}, 00010_{16}, 00020_{16}, …, $FFFE0_{16}$, $FFFF0_{16}$. Физический адрес байта памяти (20 бит) определяется суммой значения, заданного в сегментном регистре (16 бит) и умноженного на 16, со значением смещения (16 бит).

Существуют три основных типа сегментов:

- сегмент кода (`.code`) – набор команд (`CS : IP`);
- сегмент данных (`.data`) – данные программы ((`DS | ES`) : (`BX | SI | DI`));
- сегмент стека (`.stack`) – временные данные (`SS : (BP | SP)`).

3. *Структура программы.*

Программа на языке ассемблера состоит из строк общего формата:

```
метка   команда/директива операнды   ; комментарий
```

Метка может быть любой комбинацией букв английского алфавита, цифр и символов «_», «$», «@», «?», при этом цифра не может быть первым символом метки, а символы «$» и «?» иногда имеют специальные значения и не рекомендуются к использованию. Регистр символов по умолчанию не различается.

Команда процессора – мнемоника, которая транслируется в исполняемый код. Директива – мнемоника, которая не приводит к появлению нового кода, а управляет работой самого ассемблера. В поле операндов располагаются требуемые командой или директивой операнды (т. е. нельзя указать операнды и не указать команду или директиву).

Метка, стоящая перед командой, всегда заканчивается символом «:» (двоеточие) и фактически представляет собой адрес данной команды в программе. Метка, стоящая перед директивой, является одним из операндов директивы и символа двоеточия не имеет.

Комментарий – текст от символа «;» до конца строки, который не анализируется ассемблером.

Метки и комментарии в строке программы могут отсутствовать.

4. *Программные вызовы прерываний.*

Прерывание – это системная функция, которая обычно обеспечивает доступ к ресурсам компьютера и операционной системы (работа с вводом-выводом данных, переключение режимов, завершение программы и т. п.). Прерывания делятся на BIOS-прерывания (низкий уровень) и DOS-прерывания (высокий уровень). Для вызова прерывания внутри программы используется команда `INT число`.

5. *Расположение директив и форматы исполняемых файлов.*

Основные два формата исполняемых 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 программы располагаются в одном сегменте и не могут превышать 64 Кбайт.

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. Размер *exe*-файла может быть значительно больше *com*-файла за счет использования нескольких сегментов памяти.

Ниже приведены два примера программ «Hello, world!» с использованием упрощенных директив описания сегментов, поясненных в комментариях, кото-

рые демонстрируют общую структуру программ, написанных полностью на языке ассемблера.

Пример программы типа *com*:

```
; hello1.asm
; выводит на экран сообщение "Hello World!"
        .model  tiny            ; модель памяти для COM
        .code                   ; начало сегмента кода
        org 100h                ; начальное значение IP = 100h
start:  mov ah,9                ; номер функции DOS – в AH
        mov dx,offset message   ; адрес строки – в DX
        int 21h                 ; вызов системной функции DOS
        ret                     ; завершение COM-программы
message db "Hello World!",0Dh,0Ah,'$' ; строка для вывода
        end start               ; конец программы
```

Для формирования *com*-программы в среде программирования TASM используйте команды

```
tasm.exe hello1.asm hello1.obj
tlink.exe /t hello1.obj
```

Пример программы типа *exe*:

```
; hello2.asm
; выводит на экран сообщение "Hello World!"
        .model small            ; модель памяти для EXE
        .stack 100h             ; сегмент стека размером в 256 байт
        .code                   ; сегмент кода
start:  mov ax,DGROUP           ; адрес сегмента данных
        mov ds,ax               ; помещается в DS
        mov dx,offset message
        mov ah,9
        int 21h                 ; функция DOS «вывод строки»
        mov ax,4C00h
        int 21h                 ; функция DOS «завершить программу»
        .data                   ; сегмент данных
message db "Hello World!",0Dh,0Ah,'$'
        end start
```

Для формирования *exe*-программы в среде программирования TASM используйте команды

```
tasm.exe hello2.asm hello2.obj
tlink.exe hello2.obj
```

Контрольные вопросы

1. Какие регистры содержит процессор Intel 8086?
2. Что представляет собой сегментная адресация памяти?
3. Что относится к основным элементам программы на языке ассемблера?
4. Каковы правила оформления ассемблерной программы типа *com*?
5. Каковы правила оформления ассемблерной программы типа *exe*?

Дополнительные требования к выполнению работы

1. При выполнении работы постараться рассмотреть как можно больше доступных программных сред для формирования и сборки ассемблерной программы.

2. Создать ассемблерные программы для двух вариантов исполняемых модулей (*com* и *exe*).

Варианты заданий

1. Написать программу «Hello, world!».
2. Написать программу для вывода на экран строки символов.
3. Написать программу для вывода на экран нескольких строк символов.
4. Написать программу для вывода на экран строки символов, который будет сопровождаться звуковым сигналом.

Лабораторная работа №2

Тема работы: обработка символьных данных.

Цель работы: ознакомиться с директивами определения данных, изучить команды пересылки данных и передачи управления, изучить строчные операции и прерывания консольного ввода-вывода высокого уровня.

Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

1. *Директивы определения данных.*

Директивы определения данных указывают ассемблеру, что в соответствующем месте программы располагается переменная, определяют тип переменной (байт, слово и т. д.), задают ее начальное значение и ставят в соответствие переменной метку, которая будет использоваться для обращения к этим данным.

Определения данных записываются в общем виде следующим образом:

```
метка D* значение
```

Здесь `D*` является определением типа и может быть задано как:

- `DB` – байт;
- `DW` – слово (2 байта);
- `DD` – двойное слово (4 байта);
- `DF` – 6 байт (для представления адреса (`FAR` указатель));
- `DQ` – 8 байт;
- `DT` – 10 байт (80-битные данные для FPU).

Для работы с символами и строками символов в данной работе достаточно типа DB. Для работы с числовыми данными (например индексами символов в строке) лучше использовать тип DW, т. к. длина строки может превышать размер в 255 символов.

Поле значения в определении переменной может содержать одно или несколько чисел, строк символов (взятых в одиночные («'») или двойные кавычки («"»)), операторов ? и DUP, разделенных запятыми.

Имя переменной будет соответствовать адресу первого из указанных значений. Если вместо точного значения указан символ «?», то такая переменная считается неинициализированной и ее значение на момент запуска программы может оказаться любым. Если нужно заполнить участок памяти повторяющимися данными, то вместо указания точного значения используется специальный оператор DUP, имеющий формат

```
счетчик DUP (значение)
```

Счетчик – число, задающее число элементов.

Директива EQU предназначена для описания констант:

```
имя EQU операнд
```

Директива требует замены указанного имени на заданный операнд. Директиву EQU можно ставить в любое место программы.

2. *Команды пересылки данных и способы адресации.*

Базовой командой пересылки данных является команда MOV:

```
MOV приемник, источник
```

Эта команда копирует содержимое источника в приемник, источник при этом не изменяется. Команда MOV действует аналогично операторам присваивания из языков высокого уровня. В качестве источника для MOV могут использоваться: число (непосредственный операнд), регистр общего назначения, сегментный регистр или переменная (операнд, находящийся в памяти). В качестве приемника для MOV могут использоваться: регистр общего назначения, сегментный регистр (кроме CS) или переменная.

Оба операнда должны быть одного и того же размера – байт, слово или двойное слово. Нельзя выполнять пересылку данных с помощью MOV из одной переменной в другую, из одного сегментного регистра в другой и нельзя помещать в сегментный регистр непосредственный операнд – эти операции выполняют только двумя командами MOV.

Также в данной работе можно использовать стек – это специальным образом организованный участок памяти, используемый для временного хранения переменных, передачи параметров вызываемым подпрограммам и для сохранения адреса возврата при вызове процедур и прерываний. Данные можно записывать и считывать только с вершины стека. Таким образом, если записать в стек числа 1, 2, 3, то при чтении они будут получаться в обратном порядке – 3, 2, 1.

Для работы со стеком используются следующие команды:

- **PUSH источник** – поместить данные в стек;
- **POP приемник** – считать данные из стека.

Способы адресации определяют формирование адреса памяти для доступа к данным. Для правильной адресации по умолчанию (без явного указания сегментного регистра) требуется следующее:

- регистр **CS** должен указывать на начало сегмента кода – команды переходов всегда используют этот сегментный регистр;
- регистр **SS** должен указывать на начало сегмента стека – если для косвенной адресации используется регистр **BP**, то это адресация к стеку;
- регистр **DS** должен указывать на начало сегмента данных – адресация к данным по умолчанию (кроме **BP**) использует этот сегментный регистр.

Основные способы адресации данных приведены в табл. 1.

Таблица 1

Основные способы адресации данных

Способ адресации	Пример	Действие
Регистровая	`MOV AX,BX`	`AX = BX`
Непосредственная	`MOV AX,4`	`AX = 4`
Прямая	`MOV AX,ES:0001` `MOV AX,var`	`AX = слово по адресу ES:0001` `AX = слово по адресу DS:var`
Косвенная (базовая или индексная)	`MOV AX,[BX]` `MOV AX,ES:[BX]`	`AX = слово по адресу DS:BX` `AX = слово по адресу ES:BX`

Относительная косвенная	`MOV AX,[BX+2]`
	`MOV AX,2[BX]`
	`MOV AX,[BX]+2`
		`AX = слово по адресу DS:(BX+2)`

По базе с индексированием	`MOV AX,[BX+SI+2]`
	`MOV AX,2[BX][SI]`
	`MOV AX,[BX][SI]+2`
	`MOV AX,[BX+2][SI]`
		`AX = слово по адресу DS:(BX+SI+2)`

Операторы адресации:

- **SEG выражение** – сегментный адрес;
- **OFFSET выражение** – смещение;
- **THIS тип** – текущий адрес (TASM и MASM);
- **тип PTR выражение** – переопределение типа;
- **SMALL выражение** – 16-битное смещение (TASM);
- **SHORT выражение** – 8-битное смещение.

Пример использования оператора **OFFSET**:

```
mov dx,offset msg  ; занести в DX смещение переменной msg
```

3. *Команды передачи управления*.

Команды передачи управления служат для организации ветвления вычислительного процесса.

Предлагается использовать следующие команды этой группы:

- безусловный переход (**JMP метка**) – переход на метку без возврата

(от текущего положения до 32 768 байт). Для перехода в диапазоне 128 байт от текущего места можно использовать команду **JMP SHORT метка**.

- условный переход (**Jcc метка**, где **cc** – условие перехода, обычно используется после команды **CMP**) – переход в зависимости от состояния флагов, которые обычно устанавливаются предыдущей арифметической или логической операцией. Флаги, проверяемые командой, кодируются в ее мнемонике (например, **JC** – переход, если установлен флаг **CF**). Сокращения «**L**» (less – меньше) и «**G**» (greater – больше) применяются для сравнения целых чисел со знаком, а «**A**» (above – над) и «**B**» (below – под) для сравнения целых чисел без знака (табл. 2).

Таблица 2

Команды условных переходов

Команда	Условие для команды **CMP**	Реальное условие сравнения
JA **JNBE**	если выше если не ниже или равно	**CF = 0 и ZF = 0**
JAE **JNB** **JNC**	если выше или равно если не ниже если нет переноса	**CF = 0**
JB **JNAE** **JC**	если ниже если не выше или равно если перенос	**CF = 1**
JBE **JNA**	если ниже или равно если не выше	**CF = 1 и ZF = 1**
JE **JZ**	если равно если нуль	**ZF = 1**
JG **JNLE**	если больше если не меньше или равно	**ZF = 0 и SF = OF**
JGE **JNL**	если больше или равно если не меньше	**SF = OF**
JL **JNGE**	если меньше если не больше или равно	**SF <> OF**
JLE **JNG**	если меньше или равно если не больше	**ZF = 1 и SF <> OF**
JNE **JNZ**	если не равно если не нуль	**ZF = 0**
JNO	если нет переполнения	**OF = 0**
JO	если есть переполнение	**OF = 1**
JNP **JPO**	если нет четности если нечетное	**PF = 0**
JP **JPE**	если есть четность если четное	**PF = 1**
JNS	если нет знака	**SF = 0**
JS	если есть знак	**SF = 1**

- переход, если **CX = 0** (**JCXZ метка**).

Для организации условных переходов достаточно часто используется команда сравнения (**CMP источник, приемник**), которая сравнивает два числа, вычитая второе из первого, но не сохраняет результат, а лишь устанавливает в соответствии с результатом флаги состояния.

Фрагмент программы, использующей сравнения чисел:

```
        ...
        MOV BH,X            ; загрузка в BH значения X
        MOV BL,Y            ; загрузка в BL значения Y
        CMP BH,BL           ; сравнение BH и BL
        JE L_EQUAL          ; если BH=BL, то переход на L_EQUAL,
        JMP L_NOT_EQUAL     ; иначе переход на L_NOT_EQUAL
L_EQUAL:
        ...                 ; обработка варианта «числа равны»
        JMP L_CONTINUE      ; продолжение программы
L_NOT_EQUAL:
        ...                 ; обработка варианта «числа не равны»
L_CONTINUE:
        ...
```

- цикл (**LOOPxx метка**) – организация циклов в программах, используя регистр **CX** в качестве счетчика цикла. Команда **LOOP** уменьшает содержимое **CX** на единицу и передает управление на указанную метку, если содержимое **CX** не равно нулю, в противном случае выполняется команда, следующая за **LOOP**. Команда **LOOPE** (цикл «пока равно») завершает цикл, если регистр **CX** достиг нуля или если не установлен флаг нуля. Команда **LOOPNE** (цикл «пока не равно») осуществляет выход из цикла, если регистр **CX** достиг нуля или установлен флаг нуля.

Фрагмент программы, использующей цикл:

```
DSEG SEGMENT
        BUF DB "0123406789"
DSEG ENDS
CSEG SEGMENT
        ASSUME CS:CSEG, DS:DSEG, SS:SSEG
START:
        ...
        MOV BX,OFFSET BUF     ; BX – начало буфера
        MOV CX,10             ; CX - длина буфера
        MOV SI,0
L1:
        MOV DL,[BX+SI]        ; DL - символ из буфера
        MOV AH,2;
        INT 21H               ; вывод символа на экран
        INC SI                ; увеличение индекса на единицу
        LOOP L1               ; цикл
        ...
CSEG ENDS
END START
```

4. *Строковые операции*.

Кроме перечисленных выше базовых команд пересылки данных для обработки строк символов можно использовать специальные строковые операции.

Каждая строковая операция представлена в процессоре двумя видами команд, различающихся по последнему символу мнемоники команды:

- B (byte) – для обработки строк состоящих из символов-байтов (как в данной лабораторной работе);

- W (word) – для обработки строк состоящих из символов-слов.

Если флаг направления DF перед выполнением команды строковой обработки установлен в нуле (выполнена команда CLD), то значение в индексном регистре автоматически увеличивается, если в единице (выполнена команда STD) – уменьшается. Индексные регистры уменьшаются или увеличиваются на единицу, если команды работают с байтами, или на два – при работе со словами.

Строковые операции обеспечивают выполнение следующих операций:

- сравнение строк (CMPS) – команда сравнивает значение элемента одной строки (DS:SI) со значением элемента второй строки (ES:DI) и устанавливает значения индексных регистров на следующие элементы строк. Сравнение происходит так же, как и по команде сравнения CMP. Результатом операции является установка флагов;

- сканирование строки (SCAS) – команда производит сравнение содержимого аккумулятора (AL или AX) с байтом или словом памяти, абсолютный адрес которого определяется парой ES:DI, после чего регистр DI устанавливается на следующий символ. Команда SCAS используется обычно для поиска в строке (ES:DI) элемента, заданного в аккумуляторе;

- пересылка строки (MOVS) – пересылает поэлементно строку DS:SI в строку ES:DI и устанавливает значения индексных регистров на следующий элемент строки;

- запись в строку (STOS) – заполняет строку, содержащуюся по адресу ES:DI, элементом из аккумулятора (AL или AX), не влияет на флаги;

- чтение из строки (LODS) – записывает в аккумулятор (AL или AX) содержимое ячейки памяти, адрес которой задается регистрами DS:SI, не влияет на флаги.

Команды строковой обработки чаще всего используются с однобайтными префиксами (префиксами повторения), которые обеспечивают многократное автоматическое повторение выполнения следующих команд:

- повторять, пока равно (REPE);
- повторять, пока нуль (REPZ);
- повторять (REP);
- повторять, пока не равно (REPNE);
- повторять, пока не нуль (REPNZ).

Префиксы повторения ставятся перед строковыми командами обязательно в той же строке. Префикс использует регистр CX как счетчик циклов. На каждом этапе цикла выполняются следующие действия:

1) если CX=0, то выход из цикла и переход к следующей команде;
2) выполнение заданной строковой операции;
3) уменьшение CX на единицу, флаги при этом не изменяются;
4) выход из цикла, если:

- условие сравнения не выполняется для SCAS или CMPS;
- префикс REPE и ZF=0 (последнее сравнение не совпало);
- префикс REPNE и ZF=1 (последнее сравнение совпало);

5) изменение значения индексных регистров в соответствии со значением флага направления DF и переход на начало цикла.

Фрагмент программы, иллюстрирующий работу со строковыми данными, приведен ниже:

```
        CLD             ; DF=0
        LEA SI,s1       ; DS:SI = начало строки s1
        LEA DI,s2       ; ES:DI = начало строки s2
        MOV CX,n        ; CX = длина строк
        REPE CMPSB      ; сравнение, пока символы равны
        JNE NotEqual    ; если s1!=s2 (ZF=0), то на метку NotEqual
        ...             ; действия, если строки равны
NotEqual:
        ...             ; действия, если строки не равны
```

5. *Прерывания ввода-вывода*.

Прерывания ввода-вывода – специальные команды передачи управления, вызывающие функции BIOS или DOS, предоставляющие сервис по работе с аппаратурой ПЭВМ.

Для организации ввода данных с клавиатуры предлагается использовать одну из ниже приведенных функций DOS:

1) Функция DOS 01h (INT 21h) – считать символ из STDIN с эхом, ожиданием и проверкой на Ctrl + Break:

Ввод: AH = 01h.

Вывод: AL = ASCII-код символа или 0.
Если AL = 0, то второй вызов этой функции возвратит в AL расширенный ASCII-код символа.

Особенности: При чтении с помощью этой функции введенный символ автоматически немедленно отображается на экране (посылается в устройство STDOUT, так что его можно перенаправить в файл). При нажатии Ctrl + C или Ctrl + Break выполняется команда INT 23h. Если нажата клавиша, не соответствующая какому-нибудь символу (стрелки, функциональные клавиши Ins, Del и т. д.), то в AL возвращается нуль и функцию надо вызвать еще один раз, чтобы получить расширенный ASCII-код.

2) Функция DOS 06h (INT 21h) – считать символ из STDIN без эха, без ожидания и без проверки на Ctrl + Break:

Ввод: AH = 06h,
DL = 0FFh.

Вывод: ZF = 1, если не была нажата клавиша, и AL = 00;
ZF = 0, если клавиша была нажата. В этом случае AL = код символа.

3) Функция DOS 07h (INT 21h) – считать символ из STDIN без эха, с ожиданием и без проверки на Ctrl + Break:

Ввод: AH = 07h.

Вывод: AL = код символа.

4) Функция DOS 08h (INT 21h) – считать символ из STDIN без эха, с ожиданием и проверкой на Ctrl + Break:

Ввод: AH = 08h.

Вывод: AL = код символа.

5) Функция DOS 0Ah (INT 21h) – считать строку символов из STDIN в буфер:

Ввод: AH = 0Ah,
DS:DX = адрес буфера.

Вывод: Буфер содержит введенную строку.

Особенности: Для вызова этой функции надо подготовить буфер, первый байт которого содержит максимальное число символов для ввода (1 – 254), а содержимое, если оно задано, может использоваться как подсказка для ввода.

При наборе строки обрабатываются клавиши Esc, F3, F5, BS, Ctrl + C/Ctrl + Break и другие, как при наборе команд DOS (т. е. Esc начинает ввод сначала, F3 восстанавливает подсказку для ввода, F5 запоминает текущую строку как подсказку, Backspace стирает предыдущий символ).

После нажатия клавиши Enter строка (включая последний символ CR (0Dh)) записывается в буфер, начиная с третьего байта. Во второй байт записывается длина реально введенной строки без учета последнего CR.

Для вывода данных на консоль предлагается использовать одну из нижеприведенных функций DOS:

1) Функция DOS 02h (INT 21h) – записать символ в STDOUT с проверкой на Ctrl + Break:

Ввод: AH = 02h,
DL = ASCII-код символа.

Вывод: AL = код последнего записанного символа (равен DL, кроме случая, когда DL = 09h (табуляция), тогда в AL возвращается 20h).

Особенности: Эта функция при выводе на экран обрабатывает некоторые управляющие символы:

- вывод символа BEL (07h) приводит к звуковому сигналу;
- вывод символа BS (08h) приводит к перемещению курсора влево на одну позицию;
- вывод символа HT (09h) приводит к выводу нескольких пробелов;

- вывод символа LF (0Ah) перемещает курсор на одну позицию вниз;

- вывод символа CR (0Dh) приводит к переходу на начало текущей строки.

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

2) Функция DOS 06h (INT 21h) – записать символ в STDOUT без проверки на Ctrl + Break:

Ввод: AH = 06h,
DL = ASCII-код символа (кроме FFh).

Вывод: AL = код записанного символа (копия DL).

Особенности: 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.

3) Функция DOS 09h (INT 21h) – записать строку в STDOUT с проверкой на Ctrl + Break:

Ввод: AH = 09h,
DS:DX = адрес строки, заканчивающейся символом $ (24h).

Вывод: AL = 24h (код последнего символа).

Особенности: Действие этой функции полностью аналогично действию функции 02h, но выводится не один символ, а целая строка.

4) Функция DOS 40h (INT 21h) – записать строку в файл или устройство:

Ввод: AH = 40h,
BX = 01 для STDOUT или 02 для STDERR,
DS:DX = адрес начала строки,
CX = длина строки.

Вывод: CF = 0,
AX = число записанных байтов.

Особенности: Эта функция предназначена для записи в файл, но если в регистр BX поместить число 01, функция 40h будет выводить данные на STDOUT, а если BX = 02 – на устройство STDERR. STDERR всегда выводит данные на экран и не перенаправляется в файлы.

На этой функции основаны используемые в С функции стандартного вывода – фактически функция С fputs() просто вызывает это прерывание, помещая свой первый аргумент в BX, адрес строки (второй аргумент) в DS:DX, длину в CX.

6. *Макросы.*

Макросом называется фрагмент программы, который подставляется в код программы всякий раз, когда ассемблер встречает его имя в тексте программы.

Макрос начинается именем и директивой **MACRO**, а заканчивается директивой **ENDM**. После директивы **MACRO** могут быть перечислены через запятую идентификаторы параметров, используемых в макросе, что делает макрос гибким средством оформления кода.

Ниже приведен макрос **print_str** с параметром **out_str**, который предназначен для вывода строки символов на экран:

```
print_str macro out_str
    mov ah,9
    mov dx,offset out_str
    int 21h
endm
```

Контрольные вопросы

1. Какие директивы предназначены для определения данных?
2. Какие команды выполняют пересылку данных? Какие существуют способы адресации?
3. Какие команды предназначены для передачи управления?
4. Какие прерывания предназначены для ввода символов с клавиатуры?
5. Какие прерывания выполняют вывод символов на экран?

Дополнительные требования к выполнению работы

1. Выделить буфер для хранения 200 символов.
2. Строку символов ввести с клавиатуры, при этом ввод строки символов может быть завершен клавишей Enter или по заполнению буфера полностью.
3. Дополнительный буфер для хранения промежуточных результатов обработки строки в памяти не выделять.
4. При использовании констант задавать их с помощью директивы **EQU**.
5. Старт программы, ввод-вывод данных и обработку ошибок оформлять выводом в консоль поясняющих строк.

Варианты заданий

1. Отсортировать символы в строке по значению ASCII кода символа.
2. Выполнить реверс слов строки.
3. Выполнить реверс символов слов строки.
4. Отсортировать слова в строке по длине слова.
5. Отсортировать слова в строке по значению ASCII кодов символов.
6. Отсортировать слова в строке по алфавиту.
7. Удалить заданное слово в строке.
8. Удалить слово в строке, стоящее перед заданным словом.
9. Удалить слово в строке, стоящее после заданного слова.
10. Заменить заданную подстроку в строке на другую заданную подстроку.

11. Заменить заданное слово в строке на другое заданное слово.
12. Вставить в строке перед заданным словом другое заданное слово.
13. Удалить в строке слова, содержащие заданный набор букв.
14. Заменить в строке слова, содержащие заданный набор букв, на другое заданное слово.
15. Вставить в строке слово «number» перед словами, состоящими только из цифр.
16. Удалить в строке слова, являющиеся числами (учесть разные форматы).
17. Заменить в строке слова, являющиеся числами, на слово «<number>».

Лабораторная работа №3

Тема работы: целочисленные арифметические операции и обработка массивов числовых данных.

Цель работы: ознакомиться с арифметическими операциями над целочисленными данными, обработкой массивов чисел, правилами оформления ассемблерных процедур.

Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера:

1. *Арифметические операции над целыми числами.*

Арифметические операции над целыми числами в двоичной арифметике выполняются с помощью следующих команд:

- сложения (**ADD**, **ADC**):

```
ADD приемник, источник
```

Команда **ADD** выполняет арифметическое сложение приемника и источника, помещает сумму в приемник, не изменяя содержимое источника. Приемник может быть регистром или переменной, источник может быть числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Команда **ADD** никак не различает числа со знаком и без знака, но, употребляя значения флагов **CF** (перенос при сложении чисел без знака), **OF** (перенос при сложении чисел со знаком) и **SF** (знак результата), можно использовать ее и для тех, и для других.

```
ADC приемник, источник
```

Команда **ADC** во всем аналогична **ADD**, кроме того, что она выполняет арифметическое сложение приемника, источника и флага **CF**. Пара команд **ADD**/**ADC** используется для сложения чисел повышенной точности.

- вычитания (**SUB**, **SBB**):

```
SUB приемник, источник
```

Команда **SUB** вычитает источник из приемника и помещает разность в приемник. Приемник может быть регистром или переменной, источник может быть числом, регистром или переменной, но нельзя использовать переменную одновременно и для источника, и для приемника. Точно так же, как и команда **ADD**, **SUB** не делает различий между числами со знаком и без знака, но флаги позволяют использовать ее как для тех, так и для других.

SBB приемник, источник

Команда **SBB** во всем аналогична **SUB**, кроме того, что она вычитает из приемника значение источника и дополнительно вычитает значение флага **CF**.

- умножения (**MUL**, **IMUL**):

MUL источник

Команда **MUL** выполняет умножение содержимого источника (регистр или переменная) и регистра **AL**, **AX**, **EAX** (в зависимости от размера источника или оператора **PTR**) и помещает результат в **AX**, **DX:AX**, **EDX:EAX** соответственно. При умножении 8-битовых операндов результат всегда помещается в регистр **AX**. При умножении 16-битовых данных результат, который может быть длиною до 32 бит, помещается в пару регистров: в регистре **DX** содержатся старшие 16 бит, а в регистре **AX** – младшие 16 бит. Если старшая половина результата (**AH**, **DX**, **EDX**) содержит только нули (результат целиком поместился в младшую половину), то флаги **CF** и **OF** устанавливаются в нуль, иначе – в единицу. Значение остальных флагов (**SF**, **ZF**, **AF** и **PF**) не определено.

IMUL ...

Команда **IMUL** выполняет умножение с учетом знака. Эта команда имеет три формы, различающиеся числом операндов:

a) **IMUL источник**: источник (регистр или переменная) умножается на **AL**, **AX** или **EAX** (в зависимости от размера операнда) и результат располагается в **AX**, **DX:AX** или **EDX:EAX** соответственно;

b) **IMUL приемник, источник**: источник (число, регистр или переменная) умножается на приемник (регистр) и результат заносится в приемник;

c) **IMUL приемник, источник1, источник2**: источник 1 (регистр или переменная) умножается на источник 2 (число) и результат заносится в приемник (регистр).

Во всех трех вариантах считается, что результат может занимать в два раза больше места, чем размер источника. В первом случае приемник автоматически оказывается достаточно большим, но во втором и третьем случаях могут произойти переполнение и потеря старших битов результата. Флаги **OF** и **CF** будут равны единице, если это произошло, и нулю, если результат умножения поместился целиком в приемник (во втором и третьем случаях) или в младшую половину приемника (в первом случае). Значения флагов **SF**, **ZF**, **AF** и **PF** после команды **IMUL** не определены.

- деления (DIV, IDIV):

DIV источник

Команда DIV выполняет целочисленное деление без знака AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток – в AH, DX или EDX соответственно. Результат всегда округляется в сторону нуля, абсолютное значение остатка всегда меньше абсолютного значения делителя.

При 8-битном источнике (байт) используется 16-битовое делимое (AX). В результате деления получается два числа: частное помещается в регистр AL, а остаток – в AH.

При 16-битовом делителе (слово) используется 32-битовое делимое (DX:AX, причем DX содержит старшую значимую часть, а регистр AX – младшую). Команда деления помещает частное в регистр AX, а остаток в DX.

Значения флагов CF, OF, SF, ZF, AF и PF после этой команды не определены, а переполнение (если частное больше того, что может быть помещено в регистр результата (255 – для байтового деления и 65 535 – для деления слов)) или деление на нуль вызывает прерывание 0h.

IDIV источник

Команда IDIV выполняет целочисленное деление со знаком AL, AX или EAX (в зависимости от размера источника) на источник (регистр или переменная) и помещает результат в AL, AX или EAX, а остаток – в AH, DX или EDX соответственно. В остальном IDIV соответствует DIV.

Ниже приведен фрагмент программы, выполняющей вычисление формулы $X = 1 - (A \cdot 2 + B \cdot C) / (D - 3)$, где параметры X, A, B, C и D – 16-битовые целые знаковые числа:

```
...
MOV AX,2     ; AX = 2
IMUL A       ; DX:AX = A * 2
MOV BX,DX
MOV CX,AX    ; BX:CX = A * 2
MOV AX,B
IMUL C       ; DX:AX = B * C
ADD AX,CX
ADC DX,BX    ; DX:AX = A * 2 + B * C
MOV CX,D
SUB CX,3     ; CX = D – 3
IDIV CX      ; AX = (A * 2 + B * C)/(D – 3)
NEG AX       ; AX = –AX
INC AX       ; AX = AX + 1
MOV X,AX     ; X = результат
...
```

2. *Логические побитовые операции над целыми числами.*

Логические побитовые операции над целыми числами выполняются с по-

мощью следующих команд:

- логическое И (**AND**):

```
AND приемник, источник
```

Команда **AND** выполняет «логическое И» над приемником (регистр или переменная) и источником (число, регистр или переменная) и помещает результат в приемник. Источник и приемник не могут быть переменными одновременно. Флаги **OF** и **CF** обнуляются, **SF**, **ZF** и **PF** устанавливаются в соответствии с результатом, **AF** – не определен.

- логическое ИЛИ (**OR**):

```
OR приемник, источник
```

Команда **OR** выполняет «логическое ИЛИ» над приемником (регистр или переменная) и источником (число, регистр или переменная) и помещает результат в приемник. Источник и приемник не могут быть переменными одновременно. Флаги **OF** и **CF** обнуляются, **SF**, **ZF** и **PF** устанавливаются в соответствии с результатом, **AF** – не определен.

- логическое исключающее ИЛИ (**XOR**):

```
XOR приемник, источник
```

Команда **AND** выполняет «логическое И» над приемником (регистр или переменная) и источником (число, регистр или переменная) и помещает результат в приемник. Источник и приемник не могут быть переменными одновременно. Команда **XOR** часто используется для обнуления регистра:

```
   XOR AX, AX   ; обнуление AX
```

- инверсия (**NOT**):

```
NOT приемник
```

Команда **NOT** выполняет инверсию битов приемника (регистр или переменная). Флаги не затрагиваются.

3. *Процедуры.*

Процедура в ассемблере – это аналог функции С, процедур и функций PASCAL и т. п. Ассемблер не накладывает на процедуры никаких ограничений – на любой адрес программы можно передать управление командой **CALL** и оно вернется к вызвавшей процедуре, как только встретится команда **RET**. Такая свобода выражения легко может приводить к трудночитаемым программам, и в язык ассемблера были включены директивы логического оформления процедур:

```
метка PROC язык тип USES регистры
        ...
        RET
метка ENDP
```

Описание операндов **PROC**:

- *метка* – название процедуры;

- *тип* может принимать значения **NEAR** и **FAR**, и если он указан, все команды **RET** в теле процедуры будут заменены соответственно на **RETN** и **RETF**. По умолчанию подразумевается, что процедура имеет тип **NEAR** в моделях памяти **TINY**, **SMALL** и **COMPACT**;

- *язык* действует аналогично такому же операнду директивы **.MODEL**, определяя взаимодействие процедуры с языками высокого уровня. В некоторых ассемблерах директива **PROC** позволяет также считать параметры, передаваемые вызывающей программой. В этом случае указание языка необходимо, т. к. различные языки высокого уровня используют разные способы передачи параметров;

- *USES регистры* – список регистров, значения которых изменяет процедура. Ассемблер помещает в начало процедуры набор команд **PUSH**, а перед командой **RET** – набор команд **POP**, так что значения перечисленных регистров будут восстановлены.

Параметры в процедуры можно передавать в регистрах, глобальных переменных, стеке, потоке кода, блоке параметров. Одна из простых передач – передача параметров через регистры:

```
        mov ax,word ptr value   ; сделать копию значения
        call procedure          ; вызвать процедуру
```

При передаче параметров в стеке для чтения параметров из стека в процедуре обычно используют не команду **POP**, а регистр **BP**, в который помещают адрес вершины стека после входа в процедуру:

```
        push parameter1 ; поместить параметр 1 в стек
        push parameter2 ; поместить параметр 2 в стек
        call procedure
        add  sp,4       ; освободить стек от параметров
        ...
procedure proc near
        push bp         ; сохранить BP
        mov bp,sp       ; BP = вершина стека
        ...             ; команды, которые могут использовать стек
        mov ax,[bp+4]   ; считать параметр 2
    ; его адрес в сегменте стека BP+4, потому что при выполнении
    ; команды CALL в стек поместили адрес возврата – 2 байта
    ; для процедуры типа NEAR (или 4 – для FAR), а потом еще и
    ; BP – 2 байта
        mov bx,[bp+6]   ; считать параметр 1
        ...             ; остальные команды
        pop bp          ; восстановить BP
        ret             ; возврат из процедуры
procedure endp
```

Для удобства ссылок на параметры, переданные в стеке, внутри функции иногда используют директивы **EQU**, чтобы не писать каждый раз точное сме-

щение параметра от начала активационной записи (т. е. от BP), например так:

```
        push X
        push Y
        push Z
        call procedure
        ...
procedure proc near
        param_z equ [bp+8]
        param_y equ [bp+6]
        param_x equ [bp+4]
        push bp
        mov bp,sp
        ...             ; команды, которые могут использовать стек
        mov ax,param_x  ; считать параметр X
        ...             ; остальные команды процедуры
        pop bp
        ret 6           ; возврат с очисткой стека в процедуре
func endp
```

Контрольные вопросы

1. Какие проблемы возникают при вводе чисел в ассемблерных программах?
2. Какие команды выполняют арифметические операции над целыми числами?
3. Что такое переполнение при арифметических операциях?
4. Какие команды выполняют логические побитовые операции над целыми числами?
5. Что такое процедуры в ассемблере?

Дополнительные требования к выполнению работы

1. Вид буфера для хранения массива и адресацию для доступа к его элементам выбрать самостоятельно.
2. Числовые данные вводятся с клавиатуры в виде строк символов (по умолчанию используется десятичная система счисления), при этом требуется производить проверку на переполнение разрядной сетки числа (по умолчанию используются знаковые 16-битные данные), для знаковых данных знак требуется хранить в представлении самого числа (в дополнительном коде).
3. При вводе числовых массивов можно указать число вводимых элементов.
4. При операциях с целыми числами требуется проверять полученный результат на возникновение ошибок и переполнений.
5. Формирование чисел с дробной частью по условию задачи выполнять в виде массива символов на основе только целочисленных арифметических операций (без использования FPU). Выполнить округление полученного числа до N-го символа дробной части.

6. Для лучшего оформления программы ввод-вывод чисел и часто повторяющиеся действия реализовать в виде процедур.

7. Старт программы, ввод-вывод данных и обработку ошибок оформлять выводом в консоли поясняющих строк.

Варианты заданий

1. Перевести число из одной системы счисления в другую (системы счисления задаются до ввода-вывода числа в диапазоне [2…16]).
2. Выполнить набор арифметических операций над двумя целыми числами, представленными в десятеричной системе счисления.
3. Выполнить набор логических побитовых операций над двумя целыми числами, представленными в десятеричной системе счисления.
4. Выполнить набор логических побитовых операций над двумя целыми числами, представленными в шестнадцатеричной системе счисления.
5. Выполнить набор операций над двумя целыми числами.
6. Ввести массив целых чисел размерностью 30 элементов. Вычислить среднее значение элементов массива.
7. Ввести массив целых чисел размерностью 30 элементов. Найти наиболее часто встречающееся число.
8. Ввести массив целых чисел размерностью 30 элементов. Нормализовать значения элементов по максимальному числу в массиве (каждый элемент массива разделить на число с максимальным значением).
9. Ввести массив целых чисел размерностью 30 элементов. Найти отношение разности максимального и минимального значений элементов к максимальному значению.
10. Ввести массив целых чисел размерностью 30 элементов. Найти медиану элементов массива.
11. Ввести массив целых чисел размерностью 30 элементов. Подсчитать число элементов, значения которых лежат в заданном диапазоне.
12. Ввести массив целых чисел размерностью 30 элементов. Построить гистограмму для заданного диапазона чисел.
13. Ввести массив целых чисел размерностью 30 элементов. Найти вид последовательности чисел (возрастающая, убывающая, случайная).
14. Ввести массив целых чисел размерностью 30 элементов. Линейно преобразовать значения элементов к указанному диапазону (растянуть или сжать с округлением до ближайшего целого числа).
15. Ввести массив целых чисел размерностью 30 элементов. Выполнить операцию над всеми элементами массива (оператор может ввести одну из доступных операций: инверсия, модуль, возведение в квадрат, вычисление обратного значения числа).
16. Ввести матрицу целых чисел размерностью 5×6 элементов. Найти суммы элементов строк.
17. Ввести матрицу целых чисел размерностью 5×6 элементов. Найти произведения элементов столбцов.

18. Ввести матрицу целых чисел размерностью 5×6 элементов. Найти номера строк с максимальным произведением элементов.

19. Ввести матрицу целых чисел размерностью 5×6 элементов. Найти номера столбцов с минимальной суммой элементов.

20. Ввести матрицу целых чисел размерностью 5×6 элементов. Отсортировать строки по возрастанию значений элементов.

Лабораторная работа №4

Тема работы: создание видеоигры.

Цель работы: ознакомиться в рамках создания видеоигры с обработкой нажатий кнопок клавиатуры, рассмотреть прямой доступ к видеопамяти с целью формирования игрового поля и информации для пользователя.

Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

1. *Прямой доступ к видеопамяти.*

Кроме использования прерываний DOS, описанных в лабораторной работе №2, программа может выводить текст на экран с помощью пересылки данных в специальную область памяти, связанную с видеоадаптером, – видеопамять. Этот вариант вывода более быстр, чем при выводе символов через прерывания, а также позволяет формировать в консоли определенные эффекты, часто не используемые в режиме вывода в позицию курсора.

В большинстве текстовых видеорежимов под видеопамять отводится специальная область памяти, начинающаяся с абсолютного адреса `B800h:0000h` и заканчивающаяся адресом `B800h:FFFFh`. Все, что программа запишет в эту область памяти, будет пересылаться в память видеоадаптера и отображаться на экране.

В текстовых режимах для хранения каждого изображенного символа используются два байта:

- байт с ASCII-кодом символа;
- байт атрибута символа (указывает цвет символа и фона, мигание).

Байт атрибута символа имеет следующий формат (биты):

- 7 – символ мигает (по умолчанию) или фон яркого цвета (если его действие было переопределено прерыванием `10h`);
- 6 – 4 – цвет фона;
- 3 – символ яркого цвета (по умолчанию) или фон мигает (если его действие было переопределено прерыванием 11h);
- 2 – 0 – цвет символа.

Кодировка цветов приведена в табл. 3.

Таким образом, по адресу `B800h:0000h` лежит байт с кодом символа,

находящимся в верхнем левом углу экрана; по адресу B800h:0001h лежит атрибут этого символа; по адресу B800h:0002h лежит код второго символа в верхней строке экрана и т. д.

Таблица 3

Коды цветов в текстовых видеорежимах

Код цвета	Обычный цвет	Яркий цвет
000	черный	темно-серый
001	синий	светло-синий
010	зеленый	светло-зеленый
011	голубой	светло-голубой
100	красный	светло-красный
101	пурпурный	светло-пурпурный
110	коричневый	желтый
111	светло-серый	белый

Фрагмент программы, использующей прямой доступ к видеопамяти:

```
        ...
        mov cx, output_line_size ; число байт в строке – в CX
        push 0B800h
        pop es                           ; адрес в видеопамяти
        mov di,word ptr start_position   ; в ES:DI
        mov si,offset output_line        ; адрес строки в DS:SI
        cld
        rep movsb                        ; скопировать строку
        ...
start_position dw 0  ; позиция символа на экране
 ; строка " 00:00 " с атрибутом символа 1Fh (белый на синем фоне)
output_line db ' ',1Fh,
            db '0',1Fh,'0',1Fh,':',1Fh
            db '0',1Fh,'0',1Fh,' ',1Fh
output_line_size equ 14
```

Для установки требуемого программе видеорежима используется прерывание 10h (видеосервис) BIOS. Видеорежимы отличаются друг от друга разрешением (для графических) и количеством строк и столбцов (для текстовых), а также количеством возможных цветов. В данной лабораторной работе использование графических режимов видеоадаптера не требуется, поэтому в описании прерываний эта информация будет опущена.

Для установки видеорежима используются:

1) Прерывание BIOS 10h, функция 00 – установить видеорежим:

Ввод: AH = 00,

AL = номер режима в младших 7 битах:

- 00 – 40×25 черно-белый текстовый режим;
- 01 – 40×25 стандартный 16-цветный текстовый режим;
- 02 – 80×25 черно-белый текстовый режим;
- 03 – 80×25 стандартный 16-цветный текстовый режим;
- 07 – 80×25 черно-белый стандартный монохромный.

Видеорежим номер 3 используется в DOS по умолчанию.

Если старший бит AL не установлен в единицу, то экран очищается.

Вывод: Обычно отсутствует, но некоторые BIOS помещают в AL 30H для текстовых режимов и 20h для графических

Следующий пример показывает установку стандартного 16-цветового текстового режима:

```
MOV AH,00  ; функция установки видеорежима
MOV AL,03  ; цветной текст 80x25 с очисткой экрана
INT 10H    ; вызвать прерывание BIOS
```

2) Прерывание BIOS 11h – конфигурация оборудования:

Ввод: AH = 11.

Вывод: AX = состояние оборудования (биты 5 и 4 указывают текущий видеорежим):
- 00 – не используется;
- 01 – 40×25 цветной режим;
- 10 – 80×25 цветной режим;
- 11 – 80×25 черно-белый режим.

Курсор не является символом из набора ASCII-кодов. Компьютер имеет собственное аппаратное обеспечение для управления видом курсора. Обычно символ курсора похож на символ подчеркивания и всегда мерцает.

Для работы с курсором используются следующие функции BIOS:

1) Прерывание BIOS 10h, функция 01 – установить размер курсора:

Ввод: AH = 01,

CH = номер верхней линии (20H - подавить курсор),

CL = номер нижней линии.

Можно установить следующий размер курсора по вертикали:
- от 0 до 13 – для монохромных и EGA мониторов;
- от 0 до 7 – для большинства цветных мониторов.

Курсор сохраняет свой вид, пока программа не изменит его.

Стандартные размеры курсора для монитора:
- монохромный – 12/13;
- цветной – 6/7.

2) Прерывание BIOS 10h, функция 02 – установить положение курсора:

Ввод: AH = 02,

BH = номер страницы,

DH = номер строки (считая от 0, установка на строку 25 делает курсор невидимым),

DL = номер столбца (считая от 0).

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 видеорежимов 2 и 3) и 0–7 (для видеорежимов 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.

3) Прерывание BIOS `10h`, функция `03` – получить положение и размер курсора (каждая страница использует собственный независимый курсор):

Ввод: `AH = 03`,
`BH` = номер страницы.

Вывод: `DH` = строка текущей позиции курсора (считая от нуля),
`DL` = столбец текущей позиции курсора (считая от нуля),
`CH` = первая строка размера курсора,
`CL` = последняя строка размера курсора.

4) Прерывание BIOS `10h`, функция `05` – установить активную страницу (доступно для цветных текстовых режимов):

Ввод: `AH = 05`,
`AL` = номер страницы (обычно нуль).

Для режима 40×25 можно устанавливать до 8 страниц (от 0 до 7), а для режима 80×25 – до 4 страниц (от 0 до 3).

Вывод символов на неактивную страницу, а затем установка ее активной позволяют формировать более естественный и плавный вывод данных.

5) Прерывание BIOS `10h`, функция `06` – прокрутка экрана вверх (вставка чистых строк снизу):

Ввод: `AH = 06`,
если `AL = 00` – прокрутка всего экрана с заполнением пробелами (очистка), иначе `AL` = число строк для прокрутки вверх,
`BH` = атрибут вставляемого символа,
`CH` = строка верхнего левого угла окна (считая от 0),
`CL` = столбец верхнего левого угла окна (считая от 0),
`DH` = строка нижнего правого угла окна (считая от 0),
`DL` = столбец нижнего правого угла окна (считая от 0).

6) Прерывание BIOS `10h`, функция `07` – прокрутка экрана вниз (вставка чистых строк сверху):

Ввод: `AH = 07`,
если `AL = 00` – прокрутка всего экрана с заполнением пробелами (очистка), иначе AL = число строк для прокрутки вниз,
`BH` = атрибут вставляемого символа,
`CH` = строка верхнего левого угла окна (считая от 0),

CL = столбец верхнего левого угла окна (считая от 0),
DH = строка нижнего правого угла окна (считая от 0),
DL = столбец нижнего правого угла окна (считая от 0).

Прерывание 10h также обеспечивает функции вывода данных на уровне BIOS:

1) Прерывание BIOS 10h, функция 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) Прерывание BIOS 10h, функция 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.
Выводит на экран любой символ, включая специальные символы (например CR).

3) Прерывание BIOS 10h, функция 0Ah – вывести символ с текущим атрибутом на экран:

Ввод: AH = 0Ah,
BH = номер страницы,
AL = ASCII-код символа,
CX = число повторений символа.
Выводит на экран любой символ, включая специальные символы (например CR).

4) Прерывание BIOS 10h, функция 0Eh – вывести символ в режиме телетайпа:

Ввод: AH = 0Eh,
BH = номер страницы,
AL = ASCII-код символа.
Символы CR (0Dh), LF (0Ah), BEL (07h) интерпретируются как управляющие символы.
Если текст при выводе выходит за пределы нижней строки, то экран прокручивается вверх.

5) Прерывание BIOS 10h, функция 13h – вывести строку символов с заданными атрибутами:

Ввод: AH = 13h,

AL = режим вывода (биты):

- 0 – переместить курсор в конец строки после вывода;
- 1 – строка содержит не только символы, но также и атрибуты, так что каждый символ описывается двумя байтами: ASCII-код и атрибут;
- 2–7 – зарезервированы,

CX = длина строки (только число символов),

BL = атрибут, если строка содержит только символы,

DH = строка, начиная с которой будет выводиться строка символов,

DL = столбец, начиная с которого будет выводиться строка символов,

ES:BP = адрес начала строки в памяти.

Символы CR (0Dh), LF (0Ah), BEL (07h) интерпретируются как управляющие символы.

2. *Обработка нажатия кнопок клавиатуры.*

Обработка нажатий на клавиатуру может производиться различными способами:

- с помощью прерываний ввода символов DOS;
- с помощью прерываний ввода символов BIOS;
- с помощью прямого доступа к буферу клавиатуры;
- с помощью доступа к портам ввода-вывода клавиатуры.

Ввод символов с помощью функций прерывания DOS 21h рассмотрен в лабораторной работе №2. По сравнению с функциями DOS прерывание BIOS 16h предоставляет больше возможностей для считывания данных и управления клавиатурой и такой доступ практически эквивалентен по производительности прямому доступу к буферу клавиатуры.

Каждой клавише на клавиатуре соответствует уникальный код, называемый скан-код. Этот код посылается клавиатурой при каждом нажатии и отпускании клавиши и обрабатывается BIOS – записывается в кольцевой буфер клавиатуры.

Функции прерывания 16h:

1) Прерывание BIOS 16h, функция 00h (10h, 20h) – чтение символа с ожиданием:

Ввод: AH = тип клавиатуры:

- 00h – 83/84-клавиши;
- 10h – 101/102-клавиши;
- 20h – 122-клавиши.

Тип клавиатуры можно определить с помощью функции 09h прерывания 16h.

Вывод: Если нажатой клавише соответствует ASCII-символ, то в **AH** возвращается код этого символа, а в **AL** – скан-код клавиши.
Если нажатой клавише соответствует расширенный ASCII-код, то в **AL** возвращается префикс скан-кода (например **E0** для серых клавиш) или 0, если префикса нет, а в **AH** – расширенный ASCII-код.

2) Прерывание BIOS **16h**, функция **01h (11h, 21h)** – проверка наличия символа в буфере:

Ввод: **AH =** тип клавиатуры:
- **01h** – 83/84-клавиши;
- **11h** – 101/102-клавиши;
- **21h** – 122-клавиши.

Вывод: Если буфер пуст, то флаг **ZF = 1**,
если в буфере присутствует символ, то флаг **ZF = 0**, а дополнительная информация содержится в регистре **AX**:
- **AL =** ASCII-код символа, 0 или префикс скан-кода;
- **AH =** скан-код нажатой клавиши или расширенный ASCII-код.

Проверяемый символ остается в буфере клавиатуры.

3) Прерывание BIOS **16h**, функция **02h (12h, 22h)** – получить состояние клавиатуры:

Ввод: **AH =** тип клавиатуры:
- **02h** – 83/84-клавиши;
- **12h** – 101/102-клавиши;
- **22h** – 122-клавиши.

Вывод: **AL =** байт состояния клавиатуры 1,
AH = байт состояния клавиатуры 2 (только для функций **12h** и **22h**).

Байт состояния клавиатуры 1 (расположен в памяти DOS по адресу **0000h:0417h** или **0040h:0017h**):
- бит 7 – Ins включена;
- бит 6 – CapsLock включена;
- бит 5 – NumLock включена;
- бит 4 – ScrollLock включена;
- бит 3 – Alt нажата (любая Alt для функции **02h** и только левая Alt для **12h**/**22h**);
- бит 2 – Ctrl нажата (любая Ctrl);
- бит 1 – левая Shift нажата;
- бит 0 – правая Shift нажата.

Байт состояния клавиатуры 2 (расположен в памяти DOS по адресу **0000h:0418h** или **0040h:0018h**):
- бит 7 – SysRq нажата;
- бит 6 – CapsLock нажата;

- бит 5 – NumLock нажата;
- бит 4 – ScrollLock нажата;
- бит 3 – правая Alt нажата;
- бит 2 – правая Ctrl нажата;
- бит 1 – левая Alt нажата;
- бит 0 – левая Ctrl нажата.

Так как эти байты расположены в памяти по фиксированному адресу, то вместо вызова прерывания удобнее просто считывать и даже перезаписывать значения этих байтов напрямую, что изменит состояние клавиатуры.

```
; программа для выключения NumLock, CapsLock и ScrollLock
        .model tiny
        .code
        org 100h                    ; COM-файл
start:
        xor ax,ax                   ; AX = 0
        mov ds,ax                   ; DS = 0
        mov byte ptr ds:0417h,al ; байт состояния клавиатуры 1 = 0
        ret                         ; выход из программы
        end start
```

4) Прерывание BIOS `16h`, функция `05` – поместить символ в буфер клавиатуры:

Ввод: `AH = 05h`,
`CH` = скан-код (можно поместить нуль вместо скан-кода, если функция, которая будет выполнять чтение из буфера, будет использовать только ASCII-код),
`CL` = ASCII-код.

Вывод: Если операция выполнена успешно, то `AL = 00h`.
Иначе, если буфер клавиатуры переполнен, то `AL = 01h`.

К буферу клавиатуры также можно обратиться напрямую – буфер находится по адресу `0000h:041Eh` и занимает 16 слов по `0000h:043Dh` включительно. Каждый символ хранится в буфере в виде слова в таком же виде, как возвращает функция `01h` прерывания `INT 16h`.

По адресу `0000h:041Ah` находится адрес (ближний), по которому будет расположен следующий введенный символ (указатель на начало буфера), а по адресу `0000h:041Ch` лежит адрес конца буфера. Так как буфер клавиатуры закольцован, то если эти адреса начала и конца буфера равны, буфер пуст.

Иногда буфер клавиатуры размещается в другой области памяти, тогда адрес его начала хранится в области данных BIOS по адресу `0480h`, а конца – по адресу `0482h`.

3. *Доступ к системным часам.*

Персональный компьютер содержит два устройства для управления процессами:

- часы реального времени (RTC) – имеют автономное питание, использу-

ются для чтения/установки текущей даты и времени, установки будильника и для вызова прерывания IRQ8 (INT 4Ah) каждую миллисекунду;

- системный таймер – используется одновременно для управления контроллером прямого доступа к памяти, для управления динамиком и как генератор импульсов, вызывающий прерывание IRQ0 (INT 8h) 18,2 раза в секунду.

Для видеоигры, создаваемой в данной лабораторной работе, указанные выше устройства лучше всего использовать на уровне функций DOS или BIOS как средство для определения текущего времени, организации задержек и формирования случайных чисел.

Управление часами RTC и внутренними часами операционной системы средствами DOS:

1) Функция DOS 2Ah (INT 21h) – считать дату:

Ввод: AH = 2Ah.

Вывод: CX = год (1980–2099),
DH = месяц,
DL = день,
AL = день недели (0 – воскресенье, 1 – понедельник и т. п.).

2) Функция DOS 2Bh (INT 21h) – установить дату:

Ввод: AH = 2Bh,
CX = год (1980–2099),
DH = месяц,
DL = день.

Вывод: Если введена несуществующая дата, то AH = FFh,
если дата установлена, то AH = 00h.

3) Функция DOS 2Ch (INT 21h) – считать время:

Ввод: AH = 2Ch.

Вывод: CH = час,
CL = минута,
DH = секунда,
DL = сотая доля секунды.

4) Функция DOS 2Dh (INT 21h) – установить время:

Ввод: AH = 2Dh,
CH = час,
CL = минута,
DH = секунда,
DL = сотая доля секунды.

Вывод: Если введено несуществующее время, то AL = FFh,
если время установлено, то AL = 00h.

BIOS позволяет управлять часами RTC напрямую:

1) Прерывание BIOS 1Ah, функция 02h – считать время RTC:

Ввод: AH = 02h

Вывод: CF = 1 - если часы не работают или попытка чтения пришлась на момент обновления,
CF = 0 - если время успешно считано, то:
- CH = час (в формате BCD);
- CL = минута (в формате BCD);
- DH = секунда (в формате BCD);
- DL = 01h - если действует летнее время;
- DL = 00h – если не действует летнее время.

2) Прерывание BIOS 1Ah, функция 03h – установить время RTC:

Ввод: AH = 03h,
CH = час (в формате BCD),
CL = минута (в формате BCD),
DH = секунда (в формате BCD),
DL = 01h – если используется летнее время,
DL = 00h – если не используется летнее время.

3) Прерывание BIOS 1Ah, функция 04h – считать дату RTC:

Ввод: AH = 04h.

Вывод: CF = 1 – если часы не работают или попытка чтения пришлась на момент обновления,
CF = 0 – если дата успешно считана, то:
- CX = год (в формате BCD, например 1998h для 1998-го года);
- DH = месяц (в формате BCD);
- DL = день (в формате BCD).

4) Прерывание BIOS 1Ah, функция 05h – установить дату RTC:

Ввод: AH = 05h,
CX = год (в формате BCD),
DH = месяц,
DL = день.

BIOS отслеживает каждый отсчет системного таймера с помощью своего обработчика прерывания IRQ0 (INT 8h) и увеличивает на единицу значение 32-битного счетчика, который располагается в памяти по адресу 0000h:046Ch, причем при переполнении этого счетчика байт по адресу 0000h:0470h увеличивается на единицу. Программа может считывать значение этого счетчика в цикле (например, просто командой MOV) и таким образом организовывать задержки (например, ждать пока счетчик не увеличится на единицу (минимальная задержка будет равна приблизительно 55 микросекундам)). Для работы со счетчиком времени в BIOS есть следующие функции:

1) Прерывание BIOS 1Ah, функция 00h – прочитать значение счетчика времени:

Ввод: AH = 00h.

Вывод: CX:DX = значение счетчика,
AL = байт переполнения счетчика.

2) Прерывание BIOS 1Ah, функция 01h – установить значение счетчика времени:

Ввод: AH = 01h,
CX:DX = значение счетчика.

Для изменения частоты работы таймера BIOS имеет специальные функции:

1) Прерывание BIOS 15h, функция 86h – формирование задержки таймера:

Ввод: AH = 86h,
CX:DX = длительность задержки в микросекундах.

Вывод: Если таймер был занят, то CF = 1,
если задержка выполнена, то CF = 0,
AL = маска, записанная обработчиком в регистр управления прерываниями.

2) Прерывание BIOS 15h, функция 83h – управление работой счетчика:

Ввод: AH = 83h,
AL = 1 – прервать счетчик,
AL = 0 – запустить счетчик:
- CX:DX = длительность задержки в микросекундах;
- ES:BX = адрес байта, старший бит которого по окончании работы счетчика будет установлен в единицу.

Вывод: Если таймер был занят, то CF = 1,
если задержка выполнена, то CF = 0,
AL = маска, записанная обработчиком в регистр управления прерываниями.

Ниже приведен пример одного из вариантов организации задержки в видеоигре.

```
        ; начальная инициализация данных
        ...
main_cycle:
        mov cx,0
        mov dx,20000            ; пауза = 20000 микросекунд
        mov ah,86h              ; функция задержки
        int 15h                 ; задержка

        mov ah,1                ; функция проверки клавиатуры
        int 16h
        jz short no_key_pressed ; клавиша не нажата
        ; клавиша нажата - проверка, нажата ли нужная клавиша:
```

```
        xor ah,ah                   ; функция чтения кода клавиши
        int 16h                     ; AH = код клавиши
        cmp ah,K1                   ; проверка нажатия на клавишу K1
        jne short not_key_K1
        ; действия по нажатию клавиши K1
not_key_K1:
        ; проверка нажатия клавиши K2
        ...
no_key_pressed:
        ; действия, если не было нажатых клавиш
        ...
        ; проверка завершения программы
        ...
        jne short exit
        jmp short main_cycle    ; продолжить основной цикл
exit:
        ; завершение программы
        ...
```

Контрольные вопросы

1. Что такое видеорежимы и как осуществляется управление ими?
2. Что такое прямой доступ к видеопамяти?
3. Как осуществляется управление курсором и вывод данных на экран с помощью прерываний BIOS?
4. Как выполняется обработка нажатий кнопок клавиатуры с помощью прерываний BIOS? Как организован буфер клавиатуры?
5. Как осуществляется доступ к системным часам?

Дополнительные требования к выполнению работы

1. Видеоигра должна иметь простую логику работы и только одно игровое поле (уровень).
2. Для работы с игровым полем использовать прямой доступ к видеопамяти в текстовом режиме (желательно 80×25 символов).
3. Для отображения объектов подобрать адекватные символы, а также установить отвечающие ситуации атрибуты – цвет, моргание.
4. Игровое поле должно также предоставлять игроку дополнительную информацию (счет, сообщения и т. п.).
5. Желательно рассмотреть работу с системными часами или таймером (с целью формирования задержек игрового цикла, а также генерации случайных чисел).

Варианты заданий

1. Игра «Змейка».

Цель: ползаем, едим случайно появляющиеся в свободных местах яблочки и растем от этого в длину; выход за границы экрана означает возврат в поле с противоположной стороны (круглый мир).

Окончание: проигрыш – укус самого себя, выигрыш – нет.

Информация: счет.

Усложнение: смена уровня (увеличение скорости).

2. Игра «Тетрис».

Цель: заполнение случайными падающими сверху вниз фигурами (например тремя видами фигур) игрового поля; игрок может сдвигать и вращать фигуру, если нижняя строка заполнена полностью, то она очищается (все сдвигается на строку вниз).

Окончание: проигрыш – нет места для новой фигуры, выигрыш – нет.

Информация: счет.

Усложнение: смена уровня (увеличение скорости), увеличение числа видов фигур.

3. Игра «Пакман».

Цель: ходим по лабиринту и собираем случайно появляющиеся яблочки, убегаем от противника (один вид лабиринта, два или три противника).

Окончание: проигрыш – столкновение с противником, выигрыш – нет.

Информация: счет.

Усложнение: добавление бонусов (атака, увеличение скорости и т. п.).

4. Игра «Ксоникс».

Цель: от игрового поля (моря) нужно отсекать куски путем плавания по морю от суши до суши (меньший кусок без наличия внутри противников становится сушей); в море плавают противники, зеркально отражаясь от суши.

Окончание: проигрыш – столкновение линии отсечения с противником, выигрыш – площадь суши более 90 %.

Информация: площадь суши.

Усложнение: смена уровня (увеличение скорости противников).

5. Игра «Гонки».

Цель: в игровом поле дорога движется сверху вниз; необходимо объезжать препятствия, случайно расположенные на дороге; дорога может случайно изгибаться.

Окончание: проигрыш – столкновение с препятствием или выход за дорогу, выигрыш – нет.

Информация: счет времени.

Усложнение: смена уровня (увеличение скорости).

6. Игра «Арканоид».

Цель: в игровом поле с помощью мяча необходимо выбивать кирпичи стенки; мяч отражается от стен и ракетки (горизонтально перемещаемой платформы).

Окончание: проигрыш – потеря мяча, выигрыш – разбить все кирпичи.

Информация: счет разбитых кирпичей.

Усложнение: добавление бонусов (выбиваются вместе с кирпичем случайным образом).

7. Игра «Танчики».

Цель: в игровом поле движется танк игрока и танки-противники; требуется добраться до штаба противника и разрушить его.

Окончание: проигрыш – попадание снаряда противника, выигрыш – разрушение штаба противника.

Информация: счет времени.

Усложнение: добавление бонусов (появляются случайно, могут быть как полезные, так и вредные).

8. Игра «Марио».

Цель: в игровом поле расположены неподвижные препятствия и движущиеся противники, которых должен обойти герой; герой может собирать монетки.

Окончание: проигрыш – столкновение с противником, выигрыш – прохождение уровня (уровень должен быть больше ширины экрана и сдвигаться по мере движения героя).

Информация: счет собранных монеток.

Усложнение: добавление полезных бонусов, которые появляются случайно.

Лабораторная работа №5

Тема работы: работа с файлами.

Цель работы: ознакомиться с основными операциями обработки файлов, получить понятие о работе с параметрами командной строки.

Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

1. *Работа с файлами.*

Для работы с файлами в данной лабораторной работе лучше всего использовать функции DOS, которые обращаются к файлу через 16-битный идентификатор (дескриптор) файла. Такой подход более прост, чем использование более старых описателей файла (37-байтного блока управления файлом FCB) или функций низкого уровня доступа к диску (прерывание BIOS `13h`).

Первые пять значений идентификаторов такого формата инициализируются системой следующим образом:

- 0 – STDIN – стандартное устройство ввода (клавиатура);
- 1 – STDOUT – стандартное устройство вывода (экран);
- 2 – STDERR – устройство вывода сообщений об ошибках (всегда экран);
- 3 – AUX – последовательный порт (COM1);
- 4 – PRN – параллельный порт (LPT1).

Работа с файлами выполняется через функции DOS в стандартном порядке:

1) создание или открытие существующего файла;

2) выполнение файловых операций чтения или записи данных;

3) закрытие файла.

Дополнительно доступны операции: удаление, поиск и управление.

Для создания или открытия существующего файла рекомендуется использовать следующие функции:

1) Функция DOS **3Ch (INT 21h)** – создать файл:

Ввод: **AH = 3Ch**,
CX = атрибут файла (биты):
- 7 – файл можно открывать разным процессам;
- 6 – не используется;
- 5 – архивный бит (единица, если файл не сохранялся);
- 4 – каталог (должен быть нуль для функции **3Ch**);
- 3 – метка тома (игнорируется функцией **3Ch**);
- 2 – системный файл;
- 1 – скрытый файл;
- 0 – файл только для чтения.

DS:DX = адрес ASCIZ-строки с полным именем файла.

Вывод: Если **CF = 1**, то ошибка операции (в **AX** код ошибки):
- **03h** – путь не найден;
- **04h** – слишком много открытых файлов;
- **05h** – доступ запрещен.

Если **CF = 0**, то операция выполнена успешно,
AX = идентификатор файла.

Особенности: Если файл уже существует, функция все равно открывает его, присваивая ему нулевую длину.

2) Функция DOS **5Bh (INT 21h)** – создать и открыть файл:

Ввод: **AH = 5Bh**,
CX = атрибут файла,
DS:DX = адрес ASCIZ-строки с полным именем файла.

Вывод: Если CF = 1, то ошибка операции (в **AX** код ошибки):
- **03h** – путь не найден;
- **04h** – слишком много открытых файлов;
- **05h** – доступ запрещен;
- **50h** – файл уже существует.

Если **CF = 0**, то операция выполнена успешно,
AX = идентификатор файла, открытого для чтения/записи в режиме совместимости.

3) Функция DOS **5Ah (INT 21h)** – создать и открыть временный файл:

Ввод: **AH = 5Ah**,
CX = атрибут файла,
DS:DX = адрес ASCIZ-строки с путем, оканчивающимся символом «\», и тринадцатью нулевыми байтами в конце.

Вывод: Если **CF = 1**, то ошибка операции (в **AX** код ошибки):

- 03h – путь не найден;
- 04h – слишком много открытых файлов;
- 05h – доступ запрещен.

Если CF = 0, то операция выполнена успешно,

AX = идентификатор файла, открытого для чтения/записи в режиме совместимости (при этом в строку по адресу DS:DX дописывается имя созданного файла).

Особенности: Функция создает файл с уникальным именем, который не является на самом деле временным, его следует специально удалять, для чего его имя и записывается в строку в DS:DX.

4) Функция DOS 3Dh (INT 21h) – открыть существующий файл:

Ввод: AH = 3Dh,

AL = режим доступа (биты):
- 0 – открыть для чтения;
- 1 – открыть для записи;
- 3 – 2 – 00 (резерв);
- 6 – 4 – режим доступа для других процессов:
а) 000 – режим совместимости;
б) 001 – все операции запрещены;
в) 010 – запись запрещена;
г) 011 – чтение запрещено;
д) 100 – запрещений нет;
- 7 – файл не наследуется порождаемыми процессами.

DS:DX = адрес ASCIZ-строки с полным именем файла.

CL = маска атрибутов файла.

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
- 02h – файл не найден;
- 03h – путь не найден;
- 04h – слишком много открытых файлов;
- 05h – доступ запрещен;
- 0Ch – неверный режим доступа.

Если CF = 0, то операция выполнена успешно,

AX = идентификатор файла.

Во всех случаях имя файла (если диск или путь отсутствуют в описании, то системой используются их текущие значения) описывается ASCIZ-строкой (строкой ASCII-символов, оканчивающейся нулем), которая, например, имеет следующий вид:

```
filename db 'c:\data\filename.txt',0
```

Для выполнения файловых операций чтения или записи данных рекомендуется использовать следующие функции:

1) Функция DOS 3Fh (INT 21h) – чтение из файла или устройства:

Ввод: AH = 3Fh,
BX = идентификатор файла,
CX = число байт для чтения,
DS:DX = адрес буфера для приема данных.

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
- 05h – доступ запрещен;
- 06h – неверный идентификатор.
Если CF = 0, то операция выполнена успешно,
AX = число считанных байт.

Особенности: Каждая следующая операция чтения, так же как и записи, начинается не с начала файла, а с того байта, на котором остановилась предыдущая операция чтения/записи.
Если при чтении из файла число фактически считанных байт в AX меньше, чем заказанное число в CX, то при чтении был достигнут конец файла.

2) Функция DOS 42h (INT 21h) – переместить указатель чтения/записи:

Ввод: AH = 42h,
BX = идентификатор файла,
CX:DX = расстояние, на которое надо переместить указатель (знаковое число),
AL = перемещение относительно:
- 0 – начала файла;
- 1 – текущей позиции;
- 2 – конца файла.

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
06h – неверный идентификатор.
Если CF = 0, то операция выполнена успешно,
CX:DX = новое значение указателя (в байтах от начала файла).

Особенности: Указатель можно установить за реальными пределами файла:
- если указатель устанавливается в отрицательное число, то следующая операция чтения/записи вызовет ошибку;
- если указатель устанавливается в положительное число, большее длины файла, следующая операция записи увеличит размер файла.
Эта функция также часто используется для определения длины файла: достаточно вызвать ее с CX = 0, DX = 0, AL = 2, и в CX:DX будет возвращена длина файла в байтах.

3) Функция DOS 40h (INT 21h) – запись в файл или устройство:

Ввод: **AH = 40h**,
BX = идентификатор файла,
CX = число байтов для записи,
DS:DX = адрес буфера с данными.

Вывод: Если **CF = 1**, то ошибка операции (в **AX** код ошибки):
- **05h** – доступ запрещен;
- **06h** – неверный идентификатор.
Если **CF = 0**, то операция выполнена успешно,
AX = число записанных байтов.

Особенности: Если при записи в файл указать **CX = 0**, то файл будет обрезан по текущему значению указателя.
При записи в файл на самом деле происходит запись в буфер DOS, данные из которого сбрасываются на диск при закрытии файла или если их количество превышает размер сектора диска.

4) Функция DOS **68h (INT 21h)** – сброс файловых буферов DOS на диск:

Ввод: **AH = 68h**,
BX = идентификатор файла.

Вывод: Если **CF = 1**, то ошибка операции (в **AX** код ошибки).
Если **CF = 0**, то операция выполнена успешно.

5) Функция DOS **0Dh (INT 21h)** – сброс всех файловых буферов на диск:

Ввод: **AH = 0Dh**.

Для закрытия файла рекомендуется использовать функцию DOS **3Eh (INT 21h)**:

Ввод: **AH = 3Eh**,
BX = идентификатор файла.

Вывод: Если **CF = 1**, то ошибка операции (в **AX** код ошибки):
06h – неверный идентификатор.
Если **CF = 0**, то операция выполнена успешно.

Особенности: Если файл был открыт для записи, то все файловые буферы сбрасываются на диск, устанавливается время модификации файла и записывается его новая длина.

Дополнительные операции с файловой системой:

1) Функция DOS **41h (INT 21h)** – удалить файл:

Ввод: **AH = 41h**,
DS:DX = адрес ASCIZ-строки с полным именем файла.

Вывод: Если **CF = 1**, то ошибка операции (в **AX** код ошибки):
- **02h** – файл не найден;

- 03h – путь не найден;
- 05h – доступ запрещен.

Если CF = 0, то операция выполнена успешно.

Особенности: Удалить файл можно только после того, как он будет закрыт, т. к. DOS будет продолжать выполнять запись в несуществующий файл, что может привести к разрушению файловой системы.

Данная функция не позволяет использовать маску имени файла (символы «*» и «?») для удаления сразу нескольких файлов (только с DOS 7.0).

2) Функция DOS 4Eh (INT 21h) – найти первый файл:

Ввод: AH = 4Eh,

AL используется при обращении к функции APPEND,

CX = атрибуты файла (биты 0 и 5 игнорируются, если бит 3 установлен, то все остальные биты игнорируются),

DS:DX = адрес ASCIZ-строки с именем файла, которое может включать путь и маску для поиска (символы «*» и «?»).

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
- 02h – файл не найден;
- 03h – путь не найден;
- 12h – неверный режим доступа.

Если CF = 0, то операция выполнена успешно и область DTA заполняется данными о найденном файле.

Особенности: Вызов этой функции заполняет данными область памяти DTA (область передачи данных), которая начинается по умолчанию со смещения 0080h от начала блока данных PSP (при запуске *com*- и *exe*-программ сегменты DS и ES содержат сегментный адрес начала PSP), но ее можно переопределить с помощью функции DOS 1Ah.

Функции поиска файлов заполняют DTA следующим образом:

+00h (байт):
- биты 0 – 6: ASCII-код буквы диска;
- бит 7: сетевой диск;

+01h (11 байт) – маска поиска (без пути);

+0CH (байт) – атрибуты для поиска;

+0Dh (слово) – порядковый номер файла в каталоге;

+0Fh (слово) – номер кластера начала внешнего каталога;

+11h (4 байта) – зарезервировано;

+15h (байт) – атрибуты найденного файла;

+16h (слово) – время создания файла в формате DOS:
- биты 15 – 11: час (0 – 23);
- биты 10 – 05: минута;

- биты 04 – 00: номер секунды, деленный на 2 (0 – 30).
+18h (слово) – дата создания файла в формате DOS:
- биты 15 – 09: год, начиная с 1980;
- биты 08 – 05: месяц;
- биты 04 – 00: день.
+1Ah (4 байта) – размер файла;
+1Eh (13 байт) – ASCIZ-имя найденного файла (с расширением).

3) Функция DOS 1Ah (INT 21h) – установить область DTA:

Ввод: AH = 1Ah,
DS:DX = адрес начала DTA (128-байтный буфер).

4) Функция DOS 4Fh (INT 21h) – найти следующий файл:

Ввод: AH = 4Fh,
DTA должна содержать данные от предыдущего вызова функции 4E или 4F.

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки).
Если CF = 0, то операция выполнена успешно и область DTA заполняется данными о следующем найденном файле.

Особенности: Для продолжения поиска следует вызывать функцию 4Fh, пока не будет возвращена ошибка.

5) Функция DOS 39h (INT 21h) – создать каталог:

Ввод: AH = 39h,
DS:DX = адрес ASCIZ-строки с путем, в котором все каталоги, кроме последнего, существуют (для версии DOS 3.3 и более ранних длина всей строки не должна превышать 64 байта).

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
- 03h – путь не найден;
- 05h – доступ запрещен.
Если CF = 0, то операция выполнена успешно.

6) Функция DOS 3Ah (INT 21h) – удалить каталог:

Ввод: AH = 3Ah,
DS:DX = адрес ASCIZ-строки с путем, последний каталог в котором будет удален (только если он пустой, не является текущим, не занят командой SUBST).

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
- 03h – путь не найден;
- 05h – доступ запрещен;
- 10h – текущий каталог нельзя удалить.
Если CF = 0, то операция выполнена успешно.

7) Функция DOS 47h (INT 21h) – определить текущий каталог:

Ввод: AH = 47h,
DL = номер диска (00h – текущий, 01h – A: и т. д.),
DS:SI = 64-байтный буфер для текущего пути (ASCIZ-строка без имени диска, первого и последнего символа «\»).

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
0Fh – указан несуществующий диск.
Если CF = 0 и AX = 0100h, то операция выполнена успешно.

8) Функция DOS 3Bh (INT 21h) – сменить каталог:

Ввод: AH = 3Bh,
DS:DX = адрес 64-байтного ASCIZ-буфера с путем, который станет текущим каталогом.

Вывод: Если CF = 1, то ошибка операции (в AX код ошибки):
03h – путь не найден.
Если CF = 0, то операция выполнена успешно.

Фрагмент кода для чтения данных из файла и вывода этих данных на экран:

```
        ...
        f_name db 'c:\data.txt',0 ; имя файла данных
        ...
        ; открытие файла
        mov dx,offset f_name ; адрес ASCIZ-строки с именем файла
        mov ah,3Dh           ; функция DOS 3Dh
        mov al,00h           ; 00 - только чтение
        int 21h              ; открыть файл
        jc exit              ; если ошибка - выход
        mov bx,ax            ; BX = идентификатор файла
        mov di,01            ; DI = идентификатор STDOUT

        ; чтение данных из файла и запись их в STDOUT
read_data:
        mov cx,1024          ; размер блока для чтения файла
        mov dx,offset buffer ; буфер для данных
        mov ah,3Fh           ; Функция DOS 3Fh
        int 21h              ; прочитать 1024 байта из файла
        jc close_file        ; если ошибка - закрыть файл
        mov cx,ax            ; CX = число прочитанных байт
        jcxz close_file      ; если CX=0 - закрыть файл
        mov ah,40h           ; Функция DOS 40h
        xchg bx,di           ; BX = 1 - устройство STDOUT
        int 21h              ; вывод данных в STDOUT
        xchg di,bx           ; BX = идентификатор файла
        jc close_file        ; если ошибка - закрыть файл
        jmp short read_data  ; вывод следующей порции данных файла

        ; закрытие файла
```

```
close_file:
        mov ah,3Eh              ; функция DOS 3Eh
        int 21h                 ; закрыть файл

        ; завершение программы
exit:
        ...
        ; буфер данных для работы с файлом
buffer:
```

2. *Работа с командной строкой.*

Передача параметров программы через командную строку при запуске программы – решение, которое позволяет построить гибкую по входным данным программу без дополнительных диалогов с пользователем.

При запуске программы DOS помещает всю командную строку (включая последний символ `0Dh`) в блок PSP запущенной программы по смещению `81h` и ее длину в байт `80h`. Длина командной строки, хранящейся в PSP, не может быть больше 126 символов (командная строка большей длины доступна начиная с DOS 4.0 в переменной окружающей среде `CMDLINE`).

При загрузке программы в начале отводимого для нее блока памяти создается структура данных PSP (префикс программного сегмента) размером 256 байт (`100h`). Затем DOS создает копию текущего окружения для загружаемой программы, помещает полный путь и имя программы в конец окружения, заполняет поля PSP, сама программа записывается в память, начиная с адреса PSP:`0100h`.

Описание полей блока PSP приведено в табл. 4.

Таблица 4

Формат блока PSP

Адрес	Блок данных	Назначение
1	2	3
`+00h`	слово = `CDh 20h`	Команда `INT 20h` – если *com*-программа завершается командой `RETN`, то управление передается на эту команду
`+02h`	слово	Сегментный адрес первого байта после области памяти, выделенной для программы
`+04h`	байт	Резерв
`+05h`	5 байт = `9Ah F0h FEh 1Dh F0h`	Команда `CALL FAR` на абсолютный адрес `000C0h`, записанная так, чтобы второй и третий байты составляли слово, равное размеру первого сегмента для *com*-программ (`FEF0h`)
`+0Ah`	4 байта	Адрес обработчика `INT 22h` (выход из программы)
`+0Eh`	4 байта	Адрес обработчика `INT 23h` (обработчик нажатия Ctrl + Break)
`+12h`	4 байта	Адрес обработчика `INT 24h` (обработчик критических ошибок)
`+16h`	слово	Сегментный адрес PSP процесса, из которого был запущен текущий
`+18h`	20 байт	JFT – список открытых идентификаторов, один байт на идентификатор, `FFh` означает конец списка
`+2Ch`	слово	Сегментный адрес копии окружения для процесса

Окончание табл. 4

1	2	3
+2Eh	2 слова	SS:SP процесса при последнем вызове INT 21h
+32h	слово	Число элементов JFT (по умолчанию 20)
+34h	4 байта	Дальний адрес JFT (по умолчанию PSP:0018)
+38h	4 байта	Дальний адрес предыдущего PSP
+3Ch	байт	Флаг, указывающий, что консоль находится в состоянии ввода 2-байтного символа
+3Dh	байт	Флаг, устанавливаемый функцией B711h прерывания 2Fh (при следующем вызове INT 21h для работы с файлом имя файла будет заменено на полное)
+3Eh	слово	Резерв
+40h	слово	Версия DOS, которую вернет функция DOS 30h (DOS 5.0+)
+42h	12 байт	Резерв
+50h	2 байта = CDh 21h	Команда INT 21h
+54h	7 байт	Область для расширения первого FCB
+5Ch	16 байт	Первый FCB, заполняемый из первого аргумента командной строки
+6Ch	16 байт	Второй FCB, заполняемый из второго аргумента командной строки
+7Ch	4 байта	Резерв
+80h	128 байт	Командная строка и область DTA по умолчанию

При запуске *com*-программы регистры устанавливаются следующим образом:

1) AL = FFh, если первый параметр командной строки содержит неправильное имя диска (например z:/something), иначе – AL = 00h;

2) AH = FFh, если второй параметр содержит неправильное имя диска, иначе AH = 00h;

3) CS = DS = ES = SS = сегментный адрес PSP;

4) SP = адрес последнего слова в сегменте (обычно FFFEh или меньше, если не хватает памяти).

При запуске *exe*-программы регистры SS:SP устанавливаются в соответствии с сегментом стека, определенным в программе.

Затем в стек помещается слово 0000h и выполняется переход на начало программы (PSP:0100h для *com*-программы, собственная точка входа для *exe*).

Ниже приводится фрагмент *com*-программы для работы с командной строкой:

```
        .model tiny
        .code
        org 80h                 ; смещение 80h от начала PSP
cmd_length db ?                 ; длина командной строки
cmd_line   db ?                 ; сама командная строка
        org 100h                ; начало программы 100h от начала PSP
start:
        cld                     ; для команд строковой обработки
        mov bp,sp               ; сохранить текущую вершину стека в BP
```

```
        mov cl,cmd_length
        cmp cl,1            ; проверка длины командной строки
        jle exit            ; выход из программы

        ; преобразовать список параметров в PSP так, чтобы
        ; каждый параметр заканчивался нулем (ASCIZ-строка)
        ; адреса всех параметров поместить в стек
        ; в переменную argc записать число параметров

        mov cx,-1           ; для команд работы со строками
        mov di,offset cmd_line ; начало командной строки в ES:DI

find_param:
        mov al,' '          ; AL = пробел
        repz scasb          ; искать не пробел
        dec di              ; DI = адрес начала параметра
        push di             ; сохранить адрес в стек
        inc word ptr argc   ; увеличить argc на 1
        mov si,di           ; SI = DI для следующей команды lodsb

scan_params:
        lodsb               ; прочитать символ из параметра
        cmp al,0Dh          ; если 0Dh - последний параметр
        je params_ended     ; параметры закончились
        cmp al,20h          ; сравнение с пробелом
        jne scan_params     ; текущий параметр не закончился

        dec si              ; SI = первый байт после параметра
        mov byte ptr [si],0 ; записать в него 0
        mov di,si           ; DI = SI для команды scasb
        inc di              ; DI = следующий после нуля символ
        jmp short find_param ; продолжить разбор командной строки

params_ended:
        dec si              ; SI = первый байт после конца
                            ;      последнего параметра
        mov byte ptr [si],0 ; записать в него 0

        ; работа с параметрами
        ...

exit:
        ret                 ; конец программы

        ; данные программы
argc dw 0 ; число параметров

        end start
```

Контрольные вопросы

1. Каким образом выполняются операции создания, открытия и закрытия файла?
2. Каким образом осуществляется работа с данными файла? Что такое указатель чтения/записи?
3. Как осуществляется поиск файлов?
4. Каким образом выполняется работа с каталогами?
5. Как осуществляется доступ к данным командной строки?

Дополнительные требования к выполнению работы

1. Параметры обработки (включая описание путей к файлам) передавать в программу через параметры командной строки.
2. Буфер для обработки данных должен быть меньше обрабатываемого файла.
3. Под словом будем понимать набор символов, ограниченный пробелами, табуляциями, границами строки; максимальный размер слова – 50 символов.
4. Алгоритм обработки должен выполнять проверку на возникновение ошибок.
5. Старт программы, ввод-вывод данных и обработку ошибок оформлять выводом в консоли поясняющих строк.
6. Для тестирования программы подготовить файл размером больше 64 Кбайт.

Варианты заданий

1. Подсчитать число всех строк в файле.
2. Подсчитать число пустых строк в файле.
3. Подсчитать число не пустых строк в файле.
4. Подсчитать число строк длиной менее указанного значения в файле.
5. Подсчитать число строк длиной более указанного значения в файле.
6. Подсчитать число строк в файле, в которых есть все указанные символы.
7. Подсчитать число строк в файле, в которых есть указанное слово.
8. Подсчитать число строк в файле, в которых нет указанного слова.
9. В выходной файл поместить только те строки входного файла, которые содержат указанное слово.
10. В выходной файл поместить только те строки входного файла, которые не содержат указанное слово.
11. В выходной файл поместить только те строки входного файла, которые содержат все указанные символы.
12. В выходной файл поместить только те строки входного файла, которые не содержат все указанные символы.
13. Заменить во всем файле заданное слово на другое заданное слово.
14. Заменить во всем файле заданную подстроку на другую заданную подстроку.

15. Инвертировать файл по строкам.
16. Удалить в строках файла все четные слова.
17. Удалить в строках файла все нечетные слова.
18. Удалить в строках файла все N-е слова.
19. Удалить в строках файла все слова, которые совпадают с заданным.
20. Удалить в файле все неинформативные (пустые) строки.
21. Удалить в файле все информативные (не пустые) строки.
22. Удалить в файле каждую N-ю строку.
23. Удалить в файле все строки, содержащие указанное слово.
24. Написать программу-просмотрщик (viewer) содержимого файла в виде текста.
25. Написать программу-просмотрщик (viewer) содержимого файла в виде шестнадцатеричных кодов.
26. Написать программу-просмотрщик (viewer) содержимого оперативной памяти с сохранением избранных фрагментов в файл.
27. Написать программу-редактор текстового файла.
28. Написать программу-редактор бинарного файла.
29. Написать программу поиска файлов в указанном каталоге по заданному шаблону имени файла.
30. Написать программу поиска файлов в указанном каталоге, которые содержат заданную подстроку.
31. Написать программу сравнения файлов по размеру.
32. Написать программу сравнения текстовых файлов по содержимому.
33. Написать программу сравнения бинарных файлов по содержимому.
34. Написать программу сравнения каталогов по содержащимся в них наборам файлов.
35. Написать программу сравнения каталогов по содержащимся в них наборам файлов, а также по содержимому одноименных файлов.

Лабораторная работа №6

Тема работы: интерфейс с языками высокого уровня. Работа с математическим сопроцессором.

Цель работы: ознакомиться с вариантами внедрения ассемблерной процедуры в программу, написанную на языке программирования C\C++, изучить архитектуру математического сопроцессора и команды работы с ним.

Теоретические сведения

Написание программы полностью на языке ассемблера допустимо только для небольших программ. На практике используют совмещенные варианты создания программ, которые требуют сочетания ассемблера и более высоких языков программирования:

- основная часть программы пишется на языке высокого уровня, а на ассемблере пишутся отдельные процедуры, которые должны осуществлять управление нижнего уровня и (или) иметь высокую производительность;

- ассемблерная программа использует библиотечные средства языков высокого уровня.

В данной лабораторной работе выполняется создание основной программы на языке C\C++, а часть, связанная с вычислениями на математическом сопроцессоре, лежит на ассемблерной процедуре.

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

1. *Соглашения об объединении программных модулей.*

Связь ассемблерных модулей с языками высокого уровня требует следующих соглашений, которые сильно зависят от применяемых компиляторов и операционной системы:

1) Согласование вызовов.

Вызов процедуры и возврат из нее в головную программу должны быть согласованы друг с другом.

В DOS вызываемая процедура может находиться:

- в том же сегменте, что и команда вызова, при этом вызов называется близким или внутрисегментным (`NEAR`), адрес возврата занимает слово и возврат из процедуры должен быть тоже близким (`RETN`);

- в другом сегменте, тогда вызов называется дальним или межсегментным (`FAR`), адрес возврата занимает двойное слово и возврат из процедуры должен быть тоже дальним (`RETF`).

Поэтому при объединении программных модулей, написанных на языках С и ассемблера, эти модули должны использовать одну и ту же модель памяти.

В Windows используется односегментная модель памяти `FLAT`, в которой все вызовы по типу являются близкими и согласование вызовов упрощается.

2) Согласование имен.

Согласование имен требуется для того, чтобы компоновщик мог собрать исполняемый модуль. Проблемы согласования имен следующие:

- автоматическое добавление в конце имени процедуры строки `@N`, где `N` – количество передаваемых в стек параметров;

- автоматическое добавление символа «_» (подчеркивание) перед именем (например, MASM генерирует подчеркивание автоматически, а TASM этого не делает);

- согласование заглавных и прописных букв (язык С автоматически различает регистр, а для TASM нужно использовать ключ `/ml`, чтобы различать прописные и заглавные буквы);

- автоматическое добавление к концу имени перегружаемой функции в С++ некоторой строки, для того чтобы эти функции различались при компоновке, – для исключения этой проблемы нужно использовать модификатор `extern "C"`.

Чтобы обеспечить доступ к глобальным переменным при объединении модулей, необходимо выполнить следующие требования:

- если процедура на языке ассемблера вызывается из программы на языке C\C++, то такая процедура в языке ассемблера должна быть описана как **PUBLIC**;

- если переменная объявлена в программе на языке ассемблера, то в программе на ассемблере она должна иметь атрибут **PUBLIC**, а в программе на C\C++ – **extern**;

- если переменная объявлена в программе на C\C++, то в программе на ассемблере она должна иметь атрибут **EXTRN**.

3) Согласование параметров.

Для стыковки ассемблерных процедур с головной программой следует знать правила передачи параметров (табл. 5).

Таблица 5

Правила формирования параметров функций в языках высокого уровня

Соглашение	Параметры	Очистка стека	Регистры
Pascal (конвенция языка Паскаль)	Слева направо	Процедура	Нет
Fastcall (быстрый или регистровый вызов)	Слева направо	Процедура	Задействованы три регистра (**EAX**,**EDX**,**ECX**), затем стек
Cdecl (конвенция C)	Справа налево	Вызывающая программа	Нет
Stdcall (стандартный вызов)	Справа налево	Процедура	Нет

Язык C использует следующие правила формирования параметров:

- параметры помещаются в стек в порядке обратном их записи в списке параметров;

- удаление параметров из стека выполняет вызывающая программа.

При этом запись вызова функции имеет вид

```
some_proc(a, b, c, d, e)
```

и превращается в следующий ассемблерный код:

```
push e
push d
push c
push b
push a
call some_proc
add sp,10   ; освободить стек от параметров
```

Вызываемая таким образом процедура может формироваться так:

```
some_proc proc
        push bp
        mov bp,sp       ; создать стековый кадр
        a equ [bp+4]    ; простой доступ к параметру
        b equ [bp+6]
```

```
        c equ [bp+8]
        d equ [bp+10]
        e equ [bp+12]
        ...             ; исполняемый код процедуры
        pop bp
        ret
some_proc endp
```

Ассемблеры поддерживают и такой формат вызова при помощи усложненной формы директивы **proc** с указанием языка C:

```
    .model small,c
some_proc proc near uses si di, a:word, b:word
        local x:word, y:word
        ...
        ...
        ret
some_proc endp
```

Функции **some_proc** соответствует следующий прототип в программе C:

```
int some_proc(int a, int b);
```

При входе в ассемблерную процедуру в стеке будут сохранены регистры **SI** и **DI** и размещены локальные переменные **x** и **y**. Доступ к этим данным организуется с помощью адресации по базе с использованием регистра **BP**. При этом нет необходимости вычислять смещения вручную, поскольку ассемблер автоматически генерирует макроподстановки типа

```
        a EQU <WORD PTR [bp+6]>
```

Поэтому в тексте программы в качестве операндов можно использовать имена локальных переменных и передаваемых параметров.

По команде **RET** автоматически генерируются команды восстановления регистров **SI**, **DI**, **BP**, **SP** и затем только выполняется возврат в вызывающую программу.

В ассемблерной процедуре можно свободно использовать регистры **AX**, **BX**, **CX**, **DX**. Остальные регистры должны быть сохранены (например в стеке), а затем восстановлены.

Возвращаемое из процедуры значение обычно передается в регистре **AX**. Если возвращаемый результат не умещается в одном регистре, то такие данные передаются через **DX:AX**, а если результат – число с плавающей запятой, то через **ST(0)**.

Ниже приведен пример, где головная программа для DOS написана на языке C и находится в файле **C_ASM.C**, а вызываемая процедура написана на языке ассемблера и находится в файле **C_ASM.ASM**.

```
// C_ASM.C
#include <stdio.h>
extern "C" void asm_proc(int *i1, int *i2, unsigned long l1);
void main(void)
```

```
{
   int i=5, j=7;
   unsigned long l=0x12345678;
   printf("i=%d; j=%d; l=%lx\n", i, j, l);
   asm_proc(&i, &j, l);
   printf("i=%d; j=%d; l=%lx\n", i, j, l);
}
```

```
; C_ASM.ASM
.model small
.code
PUBLIC C asm_proc
asm_proc PROC near
        push bp
        mov bp,sp
        push si di
        mov si,[bp+4]
        mov di,[bp+6]
        mov bx,[bp+8]
        mov ax,[bp+10]
        mov cx,[si]
        xchg cx,[di]
        mov [si],cx
        pop di si
        pop bp
        ret
asm_proc endp
end
```

Пример головной программы **WIN_ASM.CPP** и ассемблерного модуля **WIN_ASM.ASM** для Windows:

```
// WIN_ASM.CPP
#include <iostream>
using namespace std;

extern "C" int MAS_PROC (int *, int);

int main()
{
   int *mas, n, k;
   system("chcp 1251");
   system("cls");
   cout << "Введите размер массива: ";
   cin >> n;
   mas = new int[n];
   cout << "Введите элементы массива: " << endl;
   for(int i=0; i<n; i++)
   {
      cout << "mas[" << i <<"]= ";
      cin >> mas[i];
   }
```

```
   k = MAS_PROC(mas, n);
   cout << mas[1] << "*2= " << k;
   cin.get(); cin.get();
   return 0;
}

; WIN_ASM.ASM
.586
.MODEL FLAT, C
.CODE
MAS_PROC PROC C mas:dword, n:dword
        mov esi,mas
        mov eax,[esi+4]
        shl eax, 1
        ret
MAS_PROC ENDP
END
```

2. *Встроенный ассемблер.*

Встроенный ассемблер – вставка ассемблерного кода непосредственно в код программы на языке высокого уровня. Использование встроенного ассемблера позволяет создавать программы более быстро, используя небольшие фрагменты кода без выполнения вышеизложенных требований по сборке проекта.

Любую ассемблерную команду можно записать в виде

```
asm код_операции операнды ;
```

где **asm** – оператор встроенной команды ассемблера (для компиляторов C++ от Microsoft используется ключевое слово **_asm**); **код_операции** – команда языка ассемблера (например **mov**); **операнды** – операнды этой команды (например **ax**, **bx**).

Если с помощью одного слова **asm** необходимо задать много ассемблерных команд, то они заключаются в фигурные скобки. Комментарии можно записывать только в форме, принятой в языке C++.

В программе на языке C++, использующей ассемблерные команды, иногда необходимо задать директиву **#pragma inline**, которая сообщает компилятору, что программа содержит внутренний ассемблерный код, что важно при оптимизации программы.

Ниже приведен пример программы на языке C++ с использованием встроенного ассемблера.

```
#include <iostream.h>
#pragma inline

void main(void)
{
   int a=10, b=20, c;
   cout << " a= " << a << " b= " << b << "\n";
   asm mov ax,10; // в ax значение 10
   asm mul a;     // умножение ax = ax * a
```

```
    c = _AX;
    cout << "c=" << c << "\n";

    asm
    {
       mov  ax,a
       mov  bx,b
       xchg ax,bx
       mov  a,ax
       mov  b,bx
    }

    cout << " a= " << a << " b= " << b << "\n";
}
```

В командах встроенного ассемблера можно свободно использовать переменные из языка высокого уровня, т. к. они автоматически преобразуются в соответствующие выражения.

3. *Работа с математическим сопроцессором.*

В процессорах Intel операции с плавающей запятой выполняет специальный математический сопроцессор (FPU), который имеет собственные регистры и собственный набор команд.

Сопроцессор может выполнять операции с разными типами данных, в том числе и с данными с плавающей запятой (табл. 6).

Сопроцессор выполняет все вычисления в 80-битном расширенном формате, а 32- и 64-битные числа используются для обмена данными с основным процессором и памятью.

Таблица 6

Типы данных математического сопроцессора

Тип данных	Бит	Количество значащих цифр (формат числа)	Пределы
Целое слово	16	4	–32768 – 32767
Короткое целое	32	9	$-2\cdot10^{9} - 2\cdot10^{9}$
Длинное целое	64	18	$-9\cdot10^{18} - 9\cdot10^{18}$
Упакованное десятичное	80	18	–99…99 – +99...99 (18 цифр)
Короткое вещественное	32	7 (бит 31 – знак мантиссы; биты 30–23 – 8-битная экспонента + 127; биты 22–0 – 23-битная мантисса без первой цифры)	$1.18\cdot10^{-38} - 3.40\cdot10^{38}$
Длинное вещественное	64	15–16 (бит 63 – знак мантиссы; биты 62–52 – 11-битная экспонента + 1024; биты 51–0 – 52-битная мантисса без первой цифры)	$2.23\cdot10^{-308} - 1.79\cdot10^{308}$
Расширенное вещественное	80	19 (бит 79 – знак мантиссы; биты 78–64 – 15-битная экспонента + 16 383; биты 63–0 – 64-битная мантисса, где бит 63 равен единице)	$3.37\cdot10^{-4932} - 1.18\cdot10^{4932}$

В математическом сопроцессоре есть следующие регистры:

1. Регистры данных (`R0` – `R7`) – не доступны по именам, а рассматриваются как стек, вершина которого называется `ST(0)` или просто `ST`, а следующие элементы – `ST(1)` – `ST(7)`.

2. Регистр состояний SR (рис. 2).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP	TOP	TOP	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

Рис. 2. Формат регистра состояний

Важные флаги регистра состояний:

- C3 – C0 – результат выполнения предыдущей команды; используются для условных переходов;
- TOP – номер регистра данных, который в настоящий момент является вершиной стека;
- ES – общий флаг ошибки;
- SF – ошибка стека;
- UE – флаг антипереполнения;
- OE – флаг переполнения;
- ZE – флаг деления на нуль;
- IE – флаг недопустимой операции.

3. Регистр управления CR (рис. 3).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			IC	RC	RC	PC	PC			PM	UM	OM	ZM	DM	IM

Рис. 3. Формат регистра управления

Важные флаги регистра управления:

- RC – управление округлением (0 – к ближайшему числу, 1 – к отрицательной бесконечности, 2 – к положительной бесконечности, 3 – к нулю);
- PC – управление точностью результатов команд (FADD, FSUB, FSUBR, FMUL, FDIV, FDIVR и FSQRT): 0 – одинарная точность (32-битные числа), 2 – двойная точность (64-битные), 3 – расширенная точность (80-битные);
- биты 0 – 5 – маскируют соответствующие исключения (если маскирующий бит установлен, то исключения не происходит, а результат вызвавшей его команды определяется правилами для каждого исключения специально).

4. Регистр тегов TW, который описывает текущее состояние каждого регистра данных (биты 15–14 описывают регистр R7, 13–12 – R6 и т. д.): 00 – регистр содержит число, 01 – нуль, 10 – не число (бесконечность, денормализованное число, неподдерживаемое число), 11 – регистр пуст.

5. Регистры FIP и FDP содержат адрес последней выполненной команды и адрес ее операнда соответственно (используются в обработчиках исключений для анализа вызвавшей его команды).

Команды математического сопроцессора делят на следующие группы:

1. Команды управления сопроцессором:

- FINCSTP – увеличить указатель вершины стека (если TOP было равно 7, то оно обнуляется);

- **FDECSTP** – уменьшить указатель вершины стека (если **TOP** было равно 0, то оно устанавливается в 7);

- **FFREE операнд** – освободить регистр данных **ST(n)**;

- **FINIT** – инициализировать FPU;

- **FCLEX** – обнулить флаги исключений;

- **FSTCW приемник** – сохранить регистр **CR** в приемник (16-битная переменная);

- **FLDCW источник** – загрузить регистр **CR** из источника (16-битная переменная);

- **FSAVE приемник** – сохранить состояние FPU в область памяти размером 94 или 108 байт в зависимости от разрядности операндов; инициализирует FPU аналогично команде **FINIT**;

- **FRSTOR источник** – восстановить состояние FPU;

- **FWAIT** (**WAIT**) – ожидание готовности сопроцессора (эту команду можно указывать в критических ситуациях после команд FPU, чтобы убедиться, что возможные исключения будут обработаны);

- **FNOP** – отсутствие операции.

2. Команды пересылки данных:

- **FLD источник** – загрузить вещественное число в стек – помещает содержимое источника (32-, 64- или 80-битная переменная или **ST(n)**) и уменьшает **TOP** на единицу. Команда **FLD ST(0)** делает копию вершины стека;

- **FST приемник** – скопировать вещественное число из стека – копирует **ST(0)** в приемник (32- или 64-битную переменную или пустой **ST(n)**);

- **FSTP приемник** – считать вещественное число из стека – копирует **ST(0)** в приемник (32-, 64- или 80-битную переменную или пустой **ST(n)**), а затем выталкивает число из стека (помечает **ST(0)** как пустой и увеличивает **TOP** на единицу);

- **FILD источник** – загрузить целое число в стек – преобразовывает целое число со знаком из источника (16-, 32- или 64-битная переменная) в вещественный формат, помещает на вершину стека и уменьшает **TOP** на единицу;

- **FIST приемник** – скопировать целое число из стека – преобразовывает число из вершины стека в целое со знаком и записывает его в приемник (16- или 32-битная переменная);

- **FISTP приемник** – считать целое число из стека – преобразовывает число из вершины стека в целое со знаком и записывает его в приемник (16-, 32- или 64-битная переменная), а затем выталкивает число из стека;

- **FBLD источник** – загрузить десятичное число в стек – преобразовывает число BCD из источника (80-битная переменная в памяти), помещает в вершину стека и уменьшает **TOP** на единицу;

- **FBSTP приемник** – считать десятичное число из стека – преобразовывает число из вершины стека в 80-битное упакованное десятичное, записывает его в приемник (80-битная переменная) и выталкивает это число из стека;

- **FXCH приемник** – обменять местами два регистра стека – обмен местами содержимого регистра **ST(0)** и источника (регистр **ST(n)**), если операнд не указан, обменивается содержимое **ST(0)** и **ST(1)**.

3. Команды базовой арифметики:

- **FADD приемник,источник** – сложение вещественных чисел:

а) **FADD источник** – когда источником является 32- или 64-битная переменная, а приемником – **ST(0)**;

б) **FADD ST(0),ST(n)**, **FADD ST(n),ST(0)** – когда источник и приемник заданы явно в виде регистров FPU;

в) **FADD** (без операндов) – эквивалентно **FADD ST(0),ST(1)**;

- **FADDP приемник,источник** – сложение с выталкиванием из стека:

а) **FADDP ST(n),ST(0)** – когда источник и приемник заданы явно в виде регистров FPU;

б) **FADDP** (без операндов) – эквивалентно **FADDP ST(1),ST(0)**;

- **FIADD источник** – сложение целых чисел, когда источником является 16- или 32-битная переменная, содержащая целое число, а приемником – **ST(0)**.

Варианты задания операндов для нижеперечисленных команд аналогичны вышеперечисленным ситуациям задания операндов для команд сложения (с учетом выполняемых действий):

- **FSUB приемник,источник** – вычитание вещественных чисел;
- **FSUBP приемник,источник** – вычитание с выталкиванием из стека;
- **FISUB источник** – вычитание целых чисел;
- **FSUBR приемник,источник** – обратное вычитание вещественных чисел (вычитание приемника из источника);
- **FSUBRP приемник,источник** – обратное вычитание с выталкиванием;
- **FISUBR источник** – обратное вычитание целых чисел;
- **FMUL приемник,источник** – умножение вещественных чисел;
- **FMULP приемник,источник** – умножение с выталкиванием из стека;
- **FIMUL источник** – умножение целых чисел;
- **FDIV приемник,источник** – деление вещественных чисел (некоторые ассемблеры безоперандную версию команды **FDIV** выполняют как команду **FDIVP**, т. е. безоперандная мнемоника **FDIV** выполняет **ST(1)=ST(1)/ST(0)** и выталкивает из стека верхний элемент, после чего результат оказывается в **ST(0)**);
- **FDIVP приемник,источник** – деление с выталкиванием из стека;
- **FIDIV источник** – деление целых чисел;
- **FDIVR приемник,источник** – обратное деление вещественных чисел;
- **FDIVRP приемник,источник** – обратное деление с выталкиванием из стека;

- **FIDIVR источник** – обратное деление целых чисел;

- **FABS** – найти абсолютное значение **ST(0)**;

- **FCHS** – изменить знак **ST(0)**;

- **FRNDINT** – округлить **ST(0)** до целого в соответствии с режимом округления, заданным битами **RC**;

- **FSCALE** – масштабировать по степеням двойки – умножает **ST(0)** на 2 в степени **ST(1)** (значение **ST(1)** предварительно округляется в сторону нуля до целого числа) и записывает результат в **ST(0)**;

- **FXTRACT** – извлечь экспоненту и мантиссу из числа в **ST(0)** (действие, обратное **FSCALE**) – разделяет число на мантиссу и экспоненту так, что мантисса оказывается в **ST(0)**, а экспонента – в **ST(1)**;

- **FSQRT** – извлечь квадратный корень из **ST(0)** – сохраняет результат в **ST(0)**.

4. Команды сравнения (основные):

1) **FCOM источник** – сравнить вещественные числа.

2) **FCOMP источник** – сравнить и вытолкнуть из стека.

3) **FCOMPP источник** – сравнить и вытолкнуть из стека два числа.

Команды типа **FCOM** выполняют сравнение содержимого регистра **ST(0)** с источником (32- или 64-битная переменная или регистр **ST(n)**, если операнд не указан – **ST(1)**) и устанавливают флаги согласно табл. 7.

После команд сравнения с помощью команд **FSTSW AX** и **SAHF** можно перевести флаги **C3**, **C2** и **C0** во флаги **ZF**, **PF** и **CF** соответственно, после чего все условные команды могут использовать этот результат сравнения, как после команды **CMP**.

Таблица 7

Правила формирования флагов результата при сравнении

Условие сравнения	C3	C2	C0
ST(0) > источник	0	0	0
ST(0) < источник	0	0	1
ST(0) == источник	1	0	0
данные не сравнимы	1	1	1

4) **FICOM источник** – сравнить целые числа.

5) **FICOMP источник** – сравнить целые числа и вытолкнуть из стека.

Команды типа **FICOM** сравнивают содержимое регистра **ST(0)** и источника (16- или 32-битная переменная), причем считается, что источник содержит целое число:

- **FTST** – сравнить **SP(0)** с нулем и установить флаги **C3**, **C2**, **C0**;

- **FXAM** – проанализировать содержимое **ST(0)**, устанавливает флаги, как указано в табл. 8.

Таблица 8

Правила формирования флагов результата при анализе содержимого стека

Тип числа	C3	C2	C0
Неподдерживаемое	0	0	0
Не число	0	0	1
Нормальное конечное число	0	1	0
Бесконечность	0	1	1
Нуль	1	0	0
Регистр пуст	1	0	1
Денормализованное число	1	1	0

5. Трансцендентные операции выполняют операцию над числом, находящемся в `ST(0)`, и обычно сохраняют результат в этом же регистре; для всех тригонометрических команд, операнд считается заданным в радианах и не может быть больше 2^{63} или меньше минус 2^{63}:

- `FPTAN` – тангенс (`ST(0)` содержит 1, тангенс в `ST(1)`). Единица помещается в стек для того, чтобы можно было получить котангенс вызовом команды `FDIVR` сразу после `FPTAN`;
- `FPATAN` – арктангенс числа, получаемого при делении `ST(1)` на `ST(0)`;
- `F2XMI` – вычисление 2^x-1 (x – в `ST(0)` и должно быть в диапазоне [–1...+1]);
- `FYL2X` – вычисление $y\cdot\log_2(x)$ (x – в `ST(0)` и должно быть неотрицательным, y – в `ST(1)`);
- `FSIN` – синус;
- `FCOS` – косинус.

6. Команды записи констант – помещают в `ST(0)` часто используемую в вычислениях точную константу:

- `FLD1` – 1,0;
- `FLDZ` – 0,0;
- `FLDPI` – число π;
- `FLDL2E` – $\log_2(e)$;
- `FLDL2T` – $\log_2(10)$;
- `FLDLN2` – ln(2);
- `FLDLG2` – lg(2).

Особенностями использования математического сопроцессора являются:

- необходимость инициализации с помощью команды `FINIT` перед использованием;
- параллельная работа процессора Intel 8086 и сопроцессора требуют дополнительной синхронизации, т. к. оба процессора подключены к общей системной шине, например, при работе с памятью:

```
FIST I      ; скопировать число в память I
FWAIT       ; ожидать готовности сопроцессора
MOV AX,I    ; загрузить данные в центральный процессор
```

Команда **FWAIT** приостанавливает работу центрального процессора, который может загрузить данные в регистр **AX** быстрее, чем нужные данные скопируются из сопроцессора. В современных процессорах такие операции синхронизации обычно выполняются автоматически.

Пример программы С с использованием встроенных команд сопроцессора для MS Visual Studio приведен ниже.

```
// вычислить сумму функций
// F(x) = sin(x + 1) / (cos(x) + 2)
// на интервале [a, b] с шагом step
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <conio.h>

void main(void)
{
   double a = -1000, b = 1000, step = 0.0001, x, f;
   int two = 2;
   clock_t start, end;

   x = a;
   f = 0;
   start = clock();
   _asm {
      finit       // инициализация сопроцессора

      fld b       // b загрузить в стек
                  // ST(0)=b
      fld x       // x загрузить в стек
                  // ST(0)=x, ST(1)=b
loop_start:       // метка начала цикла
      fcom        // сравнить ST(0) и ST(1)
      fstsw ax    // копировать регистр состояния в AX
      and ah,01000101b // проверить (биты: 08=C0, 10=C2, 14=C3)
      jz loop_end // если x==b, то завершить цикл

      fld1        // загрузить константу 1
                  // ST(0)=1, ST(1)=x, ST(2)=b
      fadd x      // сложить ST(0) = ST(0) + x
                  // ST(0)=1+x, ST(1)=x, ST(2)=b
      fsin        // вычислить синус ST(0) = sin(ST(0))
                  // ST(0)=sin(1+x), ST(1)=x, ST(2)=b

      fld x       // x загрузить в стек
                  // ST(0)=x, ST(1)=sin(1+x), ST(2)=x, ST(3)=b
      fcos        // косинус ST(0) = cos(ST(0))
                  // ST(0)=cos(x),ST(1)=sin(1+x),ST(2)=x, ST(3)=b
      fiadd two   // целочисленно сложить ST(0) = ST(0) + two
                  // ST(0)=cos(x)+2,ST(1)=sin(1+x),ST(2)=x,ST(3)=b
```

```
      fdiv        // делить ST(1) = ST(1) / ST(0) с выталкиванием
                  // ST(0) аналогично команде fdivp ST(1),ST(0)
                  // ST(0)=((sin(1+x)/(cos(x)+2)),ST(1)=x,ST(2)=b
      fadd f      // сложить
                  // ST(0) = ((sin(1+x)/(cos(x)+2))) + f
                  // ST(1) = x, ST(2) = b
      fstp f      // переместить ST(0) в f
                  // ST(0) = x, ST(1) = b

      fadd step   // сложить
                  // ST(0) = x + step, ST(1) = b
      fst x       // скопировать ST(0) в x
                  // ST(0) = x + step, ST(1) = b
      jmp loop_start // перейти на начало цикла
loop_end:
      fwait       // синхронизировать
   }
   end = clock();
   printf("%f\n", f);
   printf("\ntime asm %f\n", (double)(end-start)/ CLK_TCK);

   x = a;
   f = 0;
   start = clock();
   while(x <= b)
   {
      f += sin(x+1) / (cos(x) + 2);
      x += step;
   };
   end = clock();
   printf("%f\n", f);
   printf("\ntime c   %f\n", (double)(end-start)/ CLK_TCK);
   getch();
}
```

Контрольные вопросы

1. Какие соглашения о согласовании вызовов процедур и согласовании имен требуется соблюдать при объединении программных модулей?
2. Какие соглашения о передаче параметров в процедуру требуется соблюдать для языка C\C++?
3. Что такое встроенный ассемблер?
4. Какие существуют форматы данных и регистры математического сопроцессора?
5. Какие команды относятся к математическому сопроцессору?

Дополнительные требования к выполнению работы

1. Головная программа, включающая описание переменных, ввод-вывод данных и вызов ассемблерной процедуры, должна быть реализована на языке

программирования C\C++ (система программирования (компилятор) выбирается студентом).

2. Реализовать основной алгоритм обработки данных на математическом сопроцессоре как процедуру на языке ассемблера. Создание тела процедуры допускается с использованием встроенного ассемблера (актуально для головной программы, написанной для Windows).

3. Алгоритм обработки данных должен выполнять проверку на возникновение ошибок.

4. Для головной программы, написанной для Windows (модель памяти FLAT), в процедуре использовать работу с расширенными регистрами процессора.

5. Старт программы, ввод-вывод данных и обработку ошибок оформлять выводом в консоли поясняющих строк.

Варианты заданий

1. Ввести массив чисел с плавающей точкой на 10 элементов. Вычислить сумму элементов массива.

2. Ввести массив чисел с плавающей точкой на 10 элементов. Вычислить среднее значение элементов массива.

3. Ввести массив чисел с плавающей точкой на 10 элементов. Найти максимальное значение элементов массива (MAX). Для каждого элемента массива вычислить $X_i = X_i / MAX$.

4. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить $X_i = 1 / X_i$.

5. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить $X_i = (X_i)^N$ (где N = 1, 2, …).

6. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить SIN(X).

7. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить: $X_i = X_i + \sum_{j=0...i-1} (X_j)$.

8. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить: {если i – нечетное, то $X_i = SIN(X_i)$; если i – четное, то $X_i = COS(X_i)$}.

9. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить: {если $X_i < 0$, то $X_i = (X_i)^2$; если $X_i > 0$, то $X_i = (X_i)^3$}.

10. Ввести массив чисел с плавающей точкой на 10 элементов. Для каждого элемента массива вычислить: $X_i = X_i + X_{N-i}$, где N – индекс максимального элемента.

Лабораторная работа №7

Тема работы: загрузка и выполнение программ; работа с памятью.

Цель работы: ознакомиться с загрузкой и выполнением программ, рассмотреть работу с памятью.

Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

1. *Управление памятью.*

При запуске программы в DOS ей выделяется вся доступная память, поэтому возможно непосредственное использование блока памяти от конца программы и практически до конца сегмента кода программы. Для загрузки же других программ из текущей программы потребуется явно выделенная свободная память и наиболее простой вариант ее получения – сокращение выделенного текущей программе блока памяти до минимума с помощью функции DOS `4Ah`.

Функция DOS `4Ah` (`INT 21h`) – изменить размер блока памяти:

Ввод: `AH = 4Ah`,
`BX` = новый размер в 16-байтных параграфах,
`ES` = сегментный адрес модифицируемого блока.

Вывод: если `CF = 1`, то есть ошибки (в `AX` – код ошибки):
- `07` – блоки управления памятью разрушены;
- `08` – не хватает памяти (в `BX` = максимальный размер, доступный для этого блока);
- `09` – `ES` содержит неверный адрес.

Если `CF = 0`, то операция выполнена успешно.

Также доступно выделение и удаление дополнительных блоков памяти:

1) Функция DOS `48h` (`INT 21h`) – выделить блок памяти:

Ввод: `AH = 48h`,
`BX` = размер блока в 16-байтных параграфах.
Эта функция с `BX = FFFFh` используется для определения размера самого большого доступного блока памяти.

Вывод: Если `CF = 1`, то есть ошибки (в `AX` – код ошибки):
- `07` – блоки управления памятью разрушены;
- `08` – не хватает памяти (`BX` = размер максимального доступного блока).

Если `CF = 0`, то операция выполнена успешно,
`AX` = сегментный адрес выделенного блока.

2) Функция DOS `49h` (`INT 21h`) – освободить блок памяти:

Ввод: `AH = 49h`

ES = сегментный адрес освобождаемого блока

Вывод: если **CF = 1**, то есть ошибки (в **AX** – код ошибки):
- **07** – блоки управления памятью разрушены;
- **09** – **ES** содержит неверный адрес.

Если **CF = 0**, то операция выполнена успешно.

2. *Загрузка и выполнение программ.*

Для загрузки и выполнения программ требуется использовать функцию DOS **4Bh** (**INT 21h**) – загрузить и выполнить программу:

Ввод: **AH = 4Bh**,
AL = подфункции:
- **AL = 00h** – загрузить и выполнить;
- **AL = 01h** – загрузить и не выполнять:
  DS:DX – адрес ASCIZ-строки с полным именем программы,
  ES:BX – адрес блока параметров EPB:
    +00h (слово) – сегментный адрес окружения, которое будет скопировано для нового процесса (или нуль, если используется текущее окружение);
    +02h (4 байта) – адрес командной строки для нового процесса;
    +06h (4 байта) – адрес первого FCB для нового процесса;
    +0Ah (4 байта) – адрес второго FCB для нового процесса;
    +0Eh (4 байта) – здесь будет записан **SS:SP** нового процесса после его завершения (только для **AL = 01**);
    +12h (4 байта) – здесь будет записан **CS:IP** (точка входа) нового процесса после его завершения (только для **AL = 01**).
- **AL = 03h** – загрузить как оверлей:
  DS:DX – адрес ASCIZ-строки с полным именем программы;
  ES:BX – адрес блока параметров:
    +00h (слово) – сегментный адрес для загрузки оверлея;
    +02h (слово) – число, которое будет использовано в командах, использующих непосредственные сегментные адреса (обычно то же самое число, что и в предыдущем поле или нуль для *com*-программ).
- **AL = 05h** – подготовиться к выполнению (DOS 5.0+):
  DS:DX – адрес следующей структуры:
    +00h (слово) – **00h**;
    +02h (слово):
      бит 0 – *exe*-программа;
      бит 1 – программа-оверлей;
    +04h (4 байта) – адрес ASCIZ-строки с именем новой программы;

+08h (слово) – сегментный адрес PSP новой программы;
+0Ah (4 байта) – точка входа новой программы;
+0Eh (4 байта) – размер программы, включая PSP.

Вывод: Если CF = 1, то произошла ошибка (в AX = код ошибки):
- 02h – файл не найден;
- 05h – доступ к файлу запрещен;
- 08h – не хватает памяти;
- 0Ah – неправильное окружение;
- 0Bh – неправильный формат.

Если CF = 0, то операция успешно выполнена:
BX и DX изменены.

Особенности: Для подфункций 00 и 01 требуется, чтобы было достаточно свободной памяти для загрузки программы, поэтому *com*-программы должны воспользоваться функцией DOS 4Ah для уменьшения размера отведенного им блока памяти до минимально необходимого.

При вызове подфункции 03 DOS загружает оверлей в память, выделенную текущим процессом, так что *exe*-программы должны убедиться, что ее достаточно.

Эта функция игнорирует расширение файла и различает *exe*- и *com*-программы по первым двум байтам заголовка («MZ» для *exe*-файлов).

Фрагмент программы, демонстрирующий, как выполнить загрузку и запуск другой программы, приведен ниже.

```
; вызов программы test.exe
        .model tiny
        .code
        .186
        org 100h                        ; com-программа
        ...
start:
        ; перемещение стека на 200h после конца программы
        mov sp,program_length+100h+200h

        ;освободить всю память после конца программы и стека
        mov ah,4Ah
stack_shift = program_length + 100h + 200h
        mov bx,stack_shift shr 4 + 1    ; размер в параграфах + 1
        int 21h

        ; заполнить поля EPB, содержащие сегментные адреса
        mov ax,cs
        mov word ptr EPB+4,ax           ; сегмент командной строки
        mov word ptr EPB+8,ax           ; сегмент первого FCB
        mov word ptr EPB+0Ch,ax         ; сегмент второго FCB
```

```
        ; вызвать программу
        mov ax,4B00h                    ; функция DOS 4Bh
        mov dx,offset program_path      ; путь к файлу
        mov bx,offset EPB               ; блок EPB
        int 21h                         ; запустить программу

        ; выход из программы
        int 20h         ; ret нельзя - стек перемещен

        ; данные программы
program_path  db "C:\test.exe",0        ; имя файла (ASCIZ-строка)
EPB           dw 0000                   ; текущее окружение
              dw offset commandline,0   ; адрес командной строки
              dw 005Ch,0,006Ch,0        ; адреса FCB программы
commandline   db 125                    ; длина командной строки
              db " /?"                  ; командная строка (3)
command_text  db 122 dup (?)            ; командная строка (122)
program_length equ $-start              ; длина программы
        end start
```

3. *Оверлейные модули.*

Оверлей – это часть исполняемой программы (обычно процедура, хотя это может быть полностью самостоятельная программа со своими сегментами данных и стека), которая по мере необходимости загружается в определенную область памяти. Различные оверлейные модули могут загружаться в одно и то же место, перекрывая предыдущий код, что позволяет экономить память, но снижает быстродействие программы при частых загрузках. Пример ассемблерного кода оверлейного модуля приведен ниже.

```
; OVERLAY1.ASM
DSEG SEGMENT ; сегмент данных оверлея
   ...
DSEG ENDS

CSEG SEGMENT PARA PUBLIC 'CODE'   ; сегмент кода оверлея
OVERLAY PROC FAR ; процедура оверлея
ASSUME CS:CSEG,DS:DSEG
        PUSH DS                 ; сохранение DS вызывающей программы
        MOV AX,DSEG             ; установка DS на данные оверлея
        MOV DS,AX
        ...                     ; тело оверлея
        POP DS                  ; восстановление DS при завершении
        RET
OVERLAY ENDP
CSEG ENDS
END
```

Пример загрузки скомпилированного оверлейного модуля в формате *exe* приведен ниже.

```
ZSEG SEGMENT        ; фиктивный сегмент для расчета размера
```

```
ZSEG ENDS
...
; в сегменте данных основной программы
OVERLAY_SEG DW ?            ; сегмент оверлея для нулевого смещения
OVERLAY_OFFSET DW ?         ; смещение оверлея в сегменте кода
CODE_SEG DW ?               ; сегмент оверлея = сегменту кода
PATH DB 'A:\OVERLAY1.EXE'   ; путь к файлу оверлея
0BLOCK DD 0                 ; 4-байтный блок параметров
...
; в сегменте кода основной программы
        ; освобождаем память
        MOV CODE_SEG,CS     ; сегмент кода
        MOV AX,ES           ; сегмент PSP
        MOV BX,ZSEG         ; сегмент конца программы
        SUB BX,AX           ; размер памяти программы
        MOV AH,4AH
        INT 21H             ; изменение размера блока памяти
        JC SETBLK_ERROR     ; проверка ошибки

        ; отводим память для оверлея
        MOV BX,100H         ; размер блока = 1000H байт
        MOV AH,48H          ; выделение блока памяти
        INT 21H             ; AX:0000 указывает на блок памяти
        JC ALLOC_ERROR      ; проверка ошибки
        MOV OVERLAY_SEG,AX  ; сохранение сегмента оверлея

        ; вычисление смещения оверлея в сегменте кода
        MOV AX,CODE_SEG     ; сегмент оверлея
        MOV BX,OVERLAY_SEG  ; сегмент кода
        SUB BX,AX           ; BX = смещение в параграфах
        MOV CL,4
        SHL BX,CL           ; сдвиг влево на 4 бита
        MOV OVERLAY_OFFSET,BX ; смещение в байтах в сегменте кода

        ; загрузка оверлея
        MOV AX,SEG BLOCK    ; ES:BX указывает на блок параметров
        MOV ES,AX
        MOV BX,OFFSET BLOCK
        MOV AX,OVERLAY_SEG
        MOV [BX],AX         ; сегмент оверлея
        MOV [BX]+2,AX       ; фактор привязки (сегмент оверлея)
        LEA DX,PATH         ; DS:DX = путь к файлу оверлея
        MOV AH,4BH          ; номер функции загрузки программы
        MOV AL,3            ; код загрузки оверлея
        INT 21H             ; выполнение загрузки оверлея
        JC LOAD_ERROR       ; проверка ошибки
        ...
        ; вызов оверлея как далекой процедуры
        CALL DWORD PTR OVERLAY_OFFSET
        ...
```

Контрольные вопросы

1. Как осуществляется управление памятью?
2. В чем выражаются особенности загрузки и выполнения программ?
3. Для каких целей выполняется создание оверлейных модулей?
4. В чем выражаются особенности загрузки и вызова оверлейных процедур?

Дополнительные требования к выполнению работы

1. Для передачи изменяемых параметров в программу использовать командную строку.

2. Алгоритм обработки данных должен выполнять проверку на возникновение ошибок.

3. Старт программы, ввод-вывод данных и обработку ошибок оформлять выводом в консоли поясняющих строк.

Варианты заданий

1. Написать программу, запускающую другую программу N раз (N – число в диапазоне [1…255]). Имя запускаемой программы задается константой.

2. Написать программу, запускающую другую программу N раз (N – число в диапазоне [1…255]). Имя запускаемой программы передается в командной строке.

3. Написать программу, запускающую другую программу N раз (N – число в диапазоне [1…255]). Имя запускаемой программы и ее параметры передаются в командной строке.

4. Написать программу, запускающую другую программу N раз (N – число в диапазоне [1…255]). Имя запускаемой программы задается в текстовом файле в K строке (K – число в диапазоне [1…255]).

5. Написать программу, запускающую другую программу с заданными параметрами. Имя запускаемой программы задается константой, а параметры программы задаются в строках текстового файла.

6. Написать программу, запускающую другие программы. Список запускаемых программ задается в строках текстового файла.

7. Написать программу, последовательно запускающую программы, которые расположены в заданном каталоге.

8. Написать программу, запускающую саму себя N раз (N – число в диапазоне [1…255]). При запуске и окончании программы выдавать номер текущей копии.

9. Написать программу для обработки строки вида «$A_1\ x_1\ A_2\ x_2\ \dots\ x_{N-1}\ A_N$», где A_i – числа (i в диапазоне [1…N]), x_i – одна из арифметических операций (для обработки арифметических операций сформировать оверлейные функции, в память одновременно может быть загружена только одна функция, загрузку функций оптимизировать).

10. Написать программу для обработки строк вида «$A_1\ x_1\ A_2\ x_2\ \dots\ x_{N-1}\ A_N$», где A_i – числа (i в диапазоне [1…N]), x_i – одна из арифметических операций (строки брать из текстового файла, для обработки ариф-

метических операций сформировать оверлейные функции, в память одновременно может быть загружена только одна функция, загрузку функций оптимизировать)).

Лабораторная работа №8

Тема работы: обработчики прерываний и резидентные программы.

Цель работы: получить понятие об обработчиках прерываний и изучить особенности создания резидентных программ.

Теоретические сведения

Для выполнения работы требуется рассмотреть следующие элементы языка ассемблера и операционной системы:

1. *Обработчики прерываний.*

Для обработки событий, особенно происходящих асинхронно по отношению к выполнению программы, лучше всего подходит механизм прерываний – особых событий в системе, которые требуют немедленной обработки.

С каждым прерыванием связано уникальное число – номер прерывания, соответствующий определенному событию, а также соответствующий обработчик этого события. Для организации связи адреса обработчика прерывания с номером прерывания используется таблица векторов прерываний – блок памяти в диапазоне адресов от `0000:0000` до `0000:03FFh`, состоящий из 256 элементов – дальних адресов обработчиков прерываний. В первом слове элемента таблицы записано смещение, а во втором – сегмент адреса обработчика прерывания. Вектор прерывания с номером 0 находится по адресу `0000:0000`, с номером 1 – по адресу `0000:0004` и т. д. Инициализация таблицы векторов прерываний выполняется сначала BIOS перед началом загрузки операционной системы, а затем DOS (табл. 9).

Таблица 9

Описание основных векторов прерываний

Вектор	Описание прерывания
1	2
`00h`	Ошибка деления. Вызывается автоматически после выполнения команд `DIV` или `IDIV`, если в результате деления происходит переполнение (например, при делении на нуль)
`01h`	Прерывание пошагового режима. Вырабатывается после выполнения каждой машинной команды, если в слове флагов установлен бит пошаговой трассировки `TF`
`02h`	Аппаратное немаскируемое прерывание. Обычно вырабатывается при ошибке четности в оперативной памяти и при запросе прерывания от сопроцессора
`03h`	Прерывание для трассировки. Генерируется при выполнении однобайтовой машинной команды с кодом `CCh`

Окончание табл. 9

1	2
04h	Переполнение. Генерируется машинной командой **INTO**, если установлен флаг переполнения **OF**. Если флаг не установлен, команда **INTO** выполняется как **NOP**
05h	Печать копии экрана. Генерируется, если пользователь нажал клавишу PrtSc
08h	IRQ0 – прерывание интервального таймера, возникает 18,2 раза в секунду
09h	IRQ1 – прерывание от клавиатуры. Генерируется, когда пользователь нажимает и отпускает клавиши
0Ah	IRQ2 – используется для каскадирования аппаратных прерываний
0Bh	IRQ3 – прерывание асинхронного порта COM2
0Ch	IRQ4 – прерывание асинхронного порта COM1
0Dh	IRQ5 – прерывание от контроллера жесткого диска (IBM PC/XT)
0Eh	IRQ6 – прерывание контроллера НГМД после завершения ввода-вывода
0Fh	IRQ7 – прерывание от параллельного адаптера. Генерируется, когда подключенный к адаптеру принтер готов к выполнению очередной операции
10h	Обслуживание видеоадаптера
11h	Определение конфигурации устройств в системе
12h	Определение размера оперативной памяти
13h	Обслуживание дисковой системы
14h	Работа с асинхронным последовательным адаптером
16h	Обслуживание клавиатуры
17h	Обслуживание принтера
1Ah	Обслуживание часов
1Bh	Обработчик прерывания, возникающего, если пользователь нажал комбинацию клавиш Ctrl + Break
1Ch	Программное прерывание. Вызывается 18,2 раза в секунду обработчиком прерывания таймера
1Fh	Указатель на графическую таблицу для символов с кодами ASCII 128-255
60h–67h	Прерывания, зарезервированные для программ пользователя
70h	IRQ8 – прерывание от часов реального времени. Вызывается при срабатывании будильника и если они установлены на генерацию периодического прерывания (1024 раза в секунду)
71h	IRQ9 – прерывание от контроллера EGA
72h	IRQ10 – зарезервировано
73h	IRQ11 – зарезервировано
74h	IRQ12 – зарезервировано
75h	IRQ13 – прерывание ошибки от арифметического сопроцессора
76h	IRQ14 – прерывание от первого контроллера жесткого диска
77h	IRQ15 – прерывание от второго контроллера жесткого диска

Аппаратные прерывания (обозначаются IRQ0-IRQ15) вырабатываются устройствами компьютера, как правило, при завершении ими операций обмена данными или при изменении состояния и обрабатываются программируемым контроллером прерываний.

Программы могут сами вызывать прерывания с помощью команды **INT** –

программные прерывания. Программные прерывания удобно использовать для организации доступа к отдельным, общим для всех программ функциям.

2. *Резидентные программы.*

Резидентной программой называют программу, остающуюся в памяти после того, как она вернула управление операционной системе, и выполняющую обработку данных при обращении к ней.

Резидентную программу можно разделить на две части: инсталлятор и обработчик.

Инсталлятор обычно выполняет следующие действия:

- проверяет возможность установки обработчика;
- устанавливает обработчик на определенный вектор прерывания;
- завершается, оставляя обработчик резидентным.

При повторном запуске инсталлятора он может быть также использован для выгрузки резидентной части из памяти с восстановлением старых обработчиков.

Для установки вектора обработчика рекомендуется использовать следующие функции DOS:

1) Функция DOS `25h` (`INT 21h`) – установить адрес обработчика прерывания:

Ввод: `AH = 25h`,
`AL` = номер прерывания,
`DS` = сегментный адрес обработчика,
`DX` = смещение обработчика в сегменте.

2) Функция DOS `35h` (`INT 21h`) – получить адрес обработчика прерывания:

Ввод: `AH = 35h`,
`AL` = номер прерывания.

Вывод: `ES` = сегментный адрес обработчика прерывания,
`BX` = смещение обработчика прерывания.

3) Функция DOS `31h` (`INT 21h`) – оставить программу резидентной:

Ввод: `AH = 31h`,
`AL` = код возврата,
`DX` = размер резидента в 16-байтных параграфах (больше `06h`), считая от начала PSP.

Обработчик резидентной программы – процедура, выполняющая необходимые действия. Эта процедура обычно заменяет стандартный обработчик программного или аппаратного прерывания.

Принято разделять резидентные программы на активные и пассивные в зависимости от того, перехватывают ли они прерывания от внешних устройств или получают управление, только если программа специально вызовет команду `INT` с нужным номером прерывания и параметрами. Кроме этого, обработчик прерывания может быть заменен и обычной (нерезидентной) программой для

поддержки специфичных действий (по окончанию такой программы должен быть восстановлен старый обработчик прерывания).

Пример простейшего обработчика прерывания, полностью заменяющего старый обработчик:

```
int_handler proc far
        ; сохранение всех модифицируемых регистров в стеке
        ...
        ; инициализация сегментных регистров
        ...
        ; выполнение необходимых действий
        ...
        ; восстановление используемых регистров из стека
        ...
        iret        ; возврат из прерывания
int_handler endp
```

При вызове команды **INT** выполняются следующие действия:

1) в стеке сохраняются регистры **FLAGS**, **CS**, **IP**; сбрасываются флаги **IF** и **TF**;

2) в регистр **CS** из таблицы векторов прерываний заносится значение сегмента обработчика прерывания, а в регистр **IP** – смещение обработчика прерывания;

3) выполняется передача управления на обработчик программного прерывания (**CS:IP**).

При вызове команды **IRET** выполняются следующие действия:

1) из стека восстанавливаются регистры **IP**, **CS**, **FLAGS**;

2) передается управление в прерванную программу на команду, находящуюся непосредственно за командой программного прерывания **INT**.

Для сохранения возможностей стандартного обработчика можно воспользоваться следующими вариантами формирования своего обработчика (адрес старого обработчика должен быть сохранен до установки нового обработчика):

- вызов стандартного обработчика в начале функции обработчика (нижеприведенные команды эквивалентны команде **INT**):

```
pushf
call old_handler
```

- вызов стандартного обработчика в конце функции обработчика:

```
jmp cs:old_handler
```

При этом образуются цепочки обработчиков прерываний, когда один вектор прерывания имеет несколько обработчиков, вызываемых друг за другом.

Для обработчиков аппаратных прерываний рекомендуется обязательно вызывать стандартный обработчик, т. к. кроме возврата из аппаратного прерывания требуется выполнить ряд действий с аппаратурой, в том числе и с контроллером прерываний. Нижеприведенный фрагмент кода предназначен для указания контроллеру прерываний, что обработка аппаратного прерывания завершена:

```
        MOV AL,20h
        OUT 20h,AL
```

Во время работы обработчика прерываний может произойти повторный вызов этого же прерывания, при этом возможны ошибки обработки. Чтобы защитить обработчик от таких ситуаций (или сделать так, чтобы он являлся повторно входимым), можно временно запретить прерывания в критических участках кода:

```
        cli                         ; запретить прерывания
        mov al,byte ptr counter
        cmp al,100
        jb  counter_ok
        sub al,100
        mov byte ptr counter,al
counter_ok:
        sti                         ; разрешить прерывания
```

Следует помнить, что, пока прерывания запрещены, система не отслеживает изменения часов реального времени, не получает данных с клавиатуры, так что прерывания надо обязательно при первой возможности разрешать.

В DOS обработчик `INT 21h` не является повторно входимым. В отличие от прерываний BIOS, обработчики которых используют стек прерванной программы, обработчик системных функций DOS записывает в `SS:SP` адрес дна одного из трех внутренних стеков DOS. Если функция была прервана аппаратным прерыванием, обработчик которого вызвал другую функцию DOS, она будет пользоваться тем же стеком, затирая все, что туда поместила прерванная функция. Когда управление вернется в прерванную функцию, в стеке окажется мусор и произойдет ошибка. Лучший выход – вообще не использовать прерывания DOS из обработчиков прерываний, но если от этого нельзя отказаться, то нужно принять необходимые меры предосторожности.

Если прерывание произошло в тот момент, когда не выполнялось никаких системных функций DOS, ими можно пользоваться. Чтобы определить, занята DOS или нет, надо сначала до установки собственных обработчиков определить адрес флага занятости DOS с помощью функции DOS `34h` (`INT 21h`) – определить адрес флага занятости DOS:

Ввод: `AH = 34h`.

Вывод: `ES:BX` = адрес однобайтного флага занятости DOS,
`ES:BX - 1` = адрес однобайтного флага критической ошибки DOS.

Теперь обработчик прерывания может проверять состояние этих флагов и, если оба флага равны нулю, разрешается свободно пользоваться функциями DOS.

Если флаг критической ошибки не нуль, никакими функциями DOS пользоваться нельзя.

Если флаг занятости DOS не нуль, можно пользоваться только функциями `01h – 0Ch`, а чтобы воспользоваться какой-нибудь другой функцией, придется отложить действия до тех пор, пока DOS не освободится. Чтобы это сде-

лать, надо сохранить номер функции и параметры в каких-нибудь переменных в памяти и установить обработчик прерывания 08h или 1Ch. Этот обработчик будет при каждом вызове проверять флаги занятости и, если DOS освободилась, вызовет функцию с номером и параметрами, оставленными в переменных в памяти. Кроме того, участок программы после проверки флага занятости критический и прерывания на этом участке должны быть запрещены.

Не все функции DOS завершаются быстро: функция чтения символа с клавиатуры может оставаться в таком состоянии до нажатия клавиши и все это время флаг занятости DOS будет установлен в единицу. В DOS предусмотрена и такая ситуация: все функции ввода символов с ожиданием вызывают INT 28h в том же цикле, в котором они опрашивают клавиатуру, так что, если установить обработчик прерывания 28h, из него можно вызывать все функции DOS, кроме 01h – 0Ch.

Функции BIOS также часто оказываются не повторно входимыми (к ним следует отнести обработчики программных прерываний 05, 08, 09, 0Bh, 0Ch, 0Dh, 0Eh, 10h, 13h, 14h, 16h, 17h). Так как нет флага занятости BIOS, можно создать его самим:

```
int10_handler proc far
        inc  cs:byte ptr int10_busy  ; увеличить флаг занятости
        pushf                        ; передать управление старому
                                     ; обработчику INT 10h,
        call cs:dword ptr old_int10  ; эмулируя команду INT,
        dec  cs:byte ptr int10_busy  ; уменьшить флаг занятости
        iret
int10_busy db 0
int10_handler endp
```

Теперь обработчики аппаратных прерываний могут пользоваться командой INT 10h только тогда, когда флаг занятости int10_busy равен нулю, и это не приведет к ошибкам, если не найдется чужой обработчик прерывания, который тоже будет обращаться к INT 10h и не будет ничего знать о нашем флаге занятости.

Ниже приведен пример обобщенной структуры резидентной программы.

```
; резидентная программа для перехвата прерывания с номером V
        .model tiny
        .code
        .386                        ; для команд pusha, popa
        org 100h                    ; размер PSP для com-программы
start:
        jmp install_handler         ; переход на инсталлятор
old_int dd 0                        ; адрес старого обработчика
...                                 ; другие данные резидентной части
; процедура обработчика прерываний
int_handler proc far
        pushf                       ; сохранение флагов
        call cs:old_int             ; вызов старого обработчика
        ; сохранение регистров в стеке
        pusha
```

```
        push ds
        push es
        ; настройка регистра ds на данные резидентной программы
        push cs
        pop  ds
        ; обработка данных
        ...
        ; восстановление регистров из стека
        pop es
        pop ds
        popa
        ; возврат из обработчика
        iret
int_handler endp

; инсталлятор обработчика прерываний
install_handler:
        ; сохранить адрес старого обработчика
        mov ah,35h                ; получить адрес обработчика
        mov ah,V                  ; V = номер вектора обработчика
        int 21h
        mov word ptr old_int,bx   ; смещение обработчика
        mov word ptr old_int+2,es ; сегмент обработчика
        ; установить новый обработчик
        mov ah,25h                ; установить адреса обработчика
        mov al,V                  ; V = номер вектора обработчика
        int 21h
        ; завершить программу и оставить резидентную часть
        mov ax,3100h              ; оставить программу резидентной
        ; DX = размер резидентной части программы в параграфах
        mov dx,(install_handler - start + 10Fh) / 16
        int 21h
end start                         ; конец программы
```

Контрольные вопросы

1. Что представляют собой векторы прерываний?
2. Для каких целей служат аппаратные прерывания?
3. Для каких целей служат программные прерывания?
4. Из каких частей состоит инсталлятор резидентной программы?
5. Из каких частей состоит обработчик резидентной программы?

Дополнительные требования к выполнению работы

1. Для передачи параметров в программу использовать командную строку.
2. Алгоритм обработки данных должен выполнять проверку на возникновение ошибок.
3. Резидентная программа должна строить обработчик перехваченного прерывания с учетом других возможных резидентных программ, связанных с этим прерыванием.

4. Вывод данных на экран в резидентной части программы рекомендуется не выполнять с использованием прерываний DOS, лучше использовать прямой доступ к видеопамяти.

5. Старт программы, ввод-вывод данных и обработку ошибок оформлять выводом в консоли поясняющих строк.

Варианты заданий

1. Написать программу вывода строк текста из файла с обработчиком прерывания Ctrl + Break (Ctrl + C). При нажатии Ctrl + C программа должна приостановить вывод данных на экран (до нажатия любой клавиши).

2. Написать программу вывода строк текста из файла с обработчиком прерывания 5. При нажатии клавиши Prt Sc программа должна записать содержимое экрана в файл print.txt.

3. Написать резидентную программу поиска слова (задается при запуске резидента) на консоли. Фрагменты экрана с найденным словом помещать в файл FXXX.txt (где XXX – номер экрана с найденным словом).

4. Написать резидентную программу проверки правильности расстановки скобок в строках консоли. Ошибки выделять цветом.

5. Написать резидентную программу «копировщик данных консоли» (grabber) в текстовый файл. Срабатывание программы должно осуществляться по заданному действию пользователя (например нажатием клавиш Ctrl + F1).

6. Написать резидентную программу «часы» (координаты вывода данных времени передать при старте программы).

7. Написать резидентную программу «будильник». Время срабатывания будильника и длительность сигнала передать при запуске программы.

8. Написать резидентную программу «хранитель экрана» (screensaver). Время срабатывания хранителя передать при запуске программы.

9. Написать резидентную программу «просмотрщик текстового файла» (viewer). Имя файла для просмотра задавать при запуске программы.

10. Написать резидентную программу «шпион клавиатуры». Программа сохраняет все нажатия на клавиши в текстовый файл.

Литература

1. Калашников, О. А. Ассемблер? Это просто! Учимся программировать / О. А. Калашников. – СПб. : БХВ-Петербург, 2011. – 336 с.
2. Юров, В. Assembler : практикум / В. Юров. – 2-е изд.– СПб. : Питер, 2007. – 400 с.
3. Зубков, С. В. Assembler для DOS, Windows и Unix / С. В. Зубков. – СПб. : Питер, 2005. – 608 с.
4. Кип, И. Язык ассемблера для процессоров Intel / И. Кип ; пер. с англ. – 3-е изд. – М. : Изд. дом «Вильямс», 2002. – 616 с.
5. Голубь, Н. Г. Искусство программирования на Ассемблере. Лекции и упражнения / Н. Г. Голубь. – СПб. : ООО «ДиаСофтЮП», 2002. – 656 с.
6. Пирогов, В. Ю. Assembler. Учебный курс / В. Ю. Пирогов. – М. : Нолидж, 2001. – 848 с.
7. Рудаков, П. И. Язык ассемблера: уроки программирования / П. И. Рудаков, К. Г. Финогенов. – М. : Диалог-Мифи, 2001. – 640 с.
8. Юров, В. Assembler: специальный справочник / В. Юров. – СПб. : Питер, 2000. – 496 с.
9. Скляров, В. А. Программирование на языке Ассемблера / В. А. Скляров. – М. : Высш. шк., 1999. – 152 с.
10. Сван, Т. Освоение Turbo Assembler / Т. Сван. – Киев : Диалектика, 1996. – 544 с.
11. Пильщиков, В. Н. Программирование на языке ассемблера IBM PC / В. Н. Пильщиков. – М. : Диалог-Мифи, 1999. – 288 с.
12. Финогенов, К. Г. Самоучитель по системным функциям MS-DOS / К. Г. Финогенов. – М. : Малип, 1995. – 382 с.
13. Фролов, А. В. Операционная система MS-DOS / А. В. Фролов, Г. В. Фролов. – М. : Диалог-Мифи, 1992. – 222 с.
14. Фролов, А. В. Аппаратное обеспечение IBM PC / А. В. Фролов, Г. В. Фролов. – М. : Диалог-Мифи, 1992. – 208 с.
15. Касаткин, А. И. Управление ресурсами / А. И. Касаткин. – Минск : Выш. шк., 1992. – 430 с.
16. Абель, П. Язык ассемблера для IBM PC и программирования / П. Абель. – М. : Высш. шк., 1992. – 477 с.
17. Скляров, В. А. Программное и лингвистическое обеспечение персональных ЭВМ. Системы общего назначения / В. А. Скляров. – Минск : Выш. шк., 1992. – 464 с.

Содержание

Св. план 2015, поз. 33

Учебное издание

Калабухов Евгений Валерьевич
Лукьянова Ирина Викторовна
Третьяков Антон Геннадьевич

КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ. ЯЗЫК ПРОГРАММИРОВАНИЯ АССЕМБЛЕР

ПОСОБИЕ

Редактор *Е. С. Чайковская*
Корректор *Е. И. Герман*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 19.04.2016. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 4,88. Уч.-изд. л. 5,0. Тираж 100 экз. Заказ 158.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6