МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ
ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ
ФИЗИКО-ТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

К ВЫПОЛНЕНИЮ И ОФОРМЛЕНИЮ ЛАБОРАТОРНЫХ РАБОТ К КУРСУ

«ОПЕРАЦИОННЫЕ СИСТЕМЫ»

ДЛЯ СТУДЕНТОВ НАПРАВЛЕНИЯ ПОДГОТОВКИ

6.050101 «КОМПЬЮТЕРНЫЕ НАУКИ»

СПЕЦИАЛЬНОСТЕЙ «СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА»,

«КОМПЬЮТЕРНЫЙ ЭКОЛОГО-ЭКОНОМИЧЕСКИЙ МОНИТОРИНГ»

КВАЛИФИКАЦИОННОГО УРОВНЯ БАКАЛАВРА

Донецк ДонНУ 2014

ББК З973.2-018.2р30
УДК 53:004(076.5)
К 731

Автор:
В. Н. Котенко, старший преподаватель кафедры компьютерных технологий

Ответственный за выпуск:
А. А. Каргин, д-р техн. наук, профессор, зав. кафедрой компьютерных технологий ДонНУ

Утверждено на заседании ученого совета
физико-технического факультета ДонНУ.
Протокол № 9 от 23.05.2014 года

Методические указания к выполнению и оформлению лабораторных работ к курсу «Операционные системы» для студентов направления подготовки 6.050101 «Компьютерные науки» квалификационного уровня «Бакалавр» / составитель: В. Н. Котенко. – Донецк: ДонНУ, 2014. – 87 с.

Методические указания предназначены для подготовки и оформления лабораторных работ к курсу «Операционные системы» студентами кафедры компьютерных технологий и кафедры физики неравновесных процессов, метрологии и экологии в соответствии с учебным планом специальности и современными требованиями по оформлению отчетов. Приводится краткое описание каждой темы и несколько вариантов заданий по теме. Имеются примеры, разъясняющие теоретический материал. Всего восемь лабораторных работ.

Методические указания составлены на основе курса «Операционные системы», читаемого на кафедре компьютерных технологий физико-технического факультета ДонНУ. Издание может быть использовано для студентов всех специальностей, изучающих операционные системы.

З973.2-018.2р30

УДК 53:004(076.5)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Выполнение лабораторных работ студентами направления подготовки 6.050101 «Компьютерные науки» направлено на формирование знаний студентов по основам современных операционных систем.

Данные методические указания служат для углубленного изучения студентами системного программирования и архитектуры современных операционных систем и предполагают наличие у студентов навыков программирования на языках Ассемблер и С. В качестве базовой учебной модели использована архитектура процессора Intel 8086. Рассмотрены основные приемы работы со всеми наиболее важными аппаратными и программными ресурсами на низком уровне. В методических указаниях использованы таблицы, схемы, графики, а также примеры программных кодов, которые разъясняют и закрепляют теоретический материал.

Методические указания построены следующим образом. Каждая из восьми лабораторных работ состоит из следующих разделов:

1) справочно-методическая информация, которую необходимо изучить студентам для выполнения лабораторной работы;

2) контрольные вопросы по теме, правильность ответов на которые проверяется преподавателем при защите студентом лабораторной работы;

3) варианты индивидуальных заданий, которые должны выполнить студенты;

4) порядок выполнения лабораторной работы с указанием особенностей выполнения задания и оформления отчета.

Оформленный отчет по лабораторной работе должен содержать:

1) название темы лабораторной работы;

2) цель лабораторной работы;

3) контрольные вопросы и ответы на них;

4) задание и распечатку исходного кода программы и результатов.

Список рекомендуемой литературы для выполнения лабораторных работ приведен в конце методических указаний.

ЛАБОРАТОРНАЯ РАБОТА № 1
ТЕМА: ЭЛЕМЕНТЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ АССЕМБЛЕРА INTEL

Цель: знакомство с основами архитектуры и операциями языка Ассемблера микропроцессора Intel.

1. Методические указания к выполнению работы

1.1 Программно-доступные элементы процессора. Регистры. Память. Адресация памяти

В модели микропроцессора Intel 8086/88 существует ряд общедоступных программных регистров, которые приведены в таблице 1.1

Таблица 1.1 – Регистры микропроцессора Intel 8086/88

Обозначение	Назначение регистра (общепринятое)	Длина (в байтах)
AX	Аккумулятор (хранение данных и адресов)	2
BX	Регистр базы (используется при адресации данных в памяти)	2
CX	Счетчик цикла	2
DX	Регистр смещения в сегменте данных	2
SP	Указатель сегмента стека	2
BP	Базовый указатель при адресации памяти	2
SI	Индекс источника для операций над строками	2
DI	Индекс приемника для операций над строками	2
IP	Программный счетчик (смещение текущей команды в сегменте кода)	2
CS	Регистр сегмент кода	2
DS	Регистр сегмента данных	2
SS	Регистр сегмента стека	2
ES	Регистр дополнительного сегмента	2
PSW	Регистр флагов	2

Примечания.

1. Более подробно о назначении регистров можно прочитать в [4].
2. Регистры AX, BX, CX, DX могут быть представлены в виде двух однобайтовых регистров, например, для регистра AX к старшему байту можно адресоваться как AH, а к младшему – AL и т.д.

Микропроцессор INTEL 8086 имеет 20 битовую адресную шину (с возможностью адресовать до 1Мб памяти), но не имеет ни одного 20 битового регистра. Поэтому физический адрес любой ячейки памяти строится на основе содержимого двух регистров – регистра сегмента (им может быть один из сегментных регистров – CS, DS, SS, ES) и регистра смещения по нижеследующей формуле:

<Физический адрес> = <Сегментный регистр> ∗16 + <Регистр смещения>

причем адрес данной ячейки памяти можно представить в виде:

<Сегмент>:<Смещение>.

Например, адрес ячейки 0040h:0010h может быть записан и таким образом: 0000h:0410h, где h – признак 16-ричной системы исчисления, так как физический адрес – 410h.

1.2. Организация системы прерываний

Прерывание (англ. interrupt) – это приостановка выполнения текущей программы в процессоре с целью выполнения другой более важной или нужной в данный момент программы или процедуры (так называемой процедуры обработки прерывания), после завершения которой, продолжается выполнение прерванной программы с точки ее прерывания.

Операционная система MS-DOS поддерживает 256 прерываний, которые нумеруются от 0h до 0ffh. Адреса процедур обработки прерываний размещаются в памяти с адреса 0000h:0000h по 4 байта на каждое прерывание (2 байта – смещение, 2 байта – сегмент):

Смещение прерывания 0	Сегмент прерывания 0	Смещение прерывания 1	Сегмент прерывания 1	.....

Первые 32 прерывания по умолчанию обрабатываются комплексом программ обработки основных операций ввода/вывода, «зашитом» в ПЗУ (системой BIOS), остальные – непосредственно операционной системой.

Все это не относится к случаю перехвата прерываний другими программами, когда обработчики прерываний находятся полностью или частично в теле этих программ.

Для *вызова* соответствующего прерывания необходимо загрузить в регистры необходимую для работы прерывания информацию. Перечень регистров строго определен для каждого прерывания. Результаты работы прерывания возвращаются в соответствующих регистрах.

Прерывания бывают 3-х типов:

1) *аппаратные* – связаны с выработкой сигналов от аппаратуры, например: падение напряжения, прерывание от клавиатуры и т.д.;

2) *логические* – связаны с обработкой нестандартных ситуаций при работе программы, как-то – деление на 0, переполнение порядка и т.д.;

3) *программные* – вызываются по инструкции INT на Ассемблере или аналогичной команде на языке высокого уровня.

В таблице 1.2 приведен перечень некоторых распространенных прерываний (номера прерываний представлены в 16-ричной форме).

Таблица 1.2 – Перечень распространенных прерываний

Номер прерывания	Функциональное назначение	Тип прерывания
00h	Деление на 0	Логическое
04h	Переполнение	Логическое
08h	Прерывание от таймера	Аппаратное
09h	Прерывание от клавиатуры	Аппаратное
10h	Видео-сервис	Программное
13h	Прерывание работы с диском	Программное
21h	Функции работы с операционной системой	Программное
33h	Функции поддержки работы манипулятора «мышь»	Аппаратное

2. Контрольные вопросы по теме

1. Дайте характеристику программно-доступных регистров микропроцессора Intel 8086/88.
2. Адресация памяти в микропроцессоре 8086/88.
3. Что такое прерывание?
4. Организация прерываний в операционной системе MS-DOS.
5. Типы прерываний.
6. Дайте характеристику наиболее распространенных прерываний.

3. Варианты индивидуальных заданий

ВАРИАНТ 1

Составить программу получения и изменения текущей даты. Используйте прерывание 21h (функции 2AH, 2BH).

1. *Получение текущей даты.*

Входные регистры (далее везде «вход»): AH = 2AH;

Выходные регистры (далее везде «выход»): DL – день (1 до 31), DH – месяц (1 до 12), CX – год (1980 до 2099), AL – день недели (0 – воскресенье, 1 – понедельник и т.д.).

2. *Установка текущей даты.*

Вход: AH = 2BH, DL – день, DH – месяц, CX – год.
Выход: AL = 00H – если дата корректна; 0FFH – если дата некорректна.

ВАРИАНТ 2

Составить программу, дублирующую функции операционной системы для работы с каталогами (создание, удаление, изменение текущего каталога). Используйте прерывание 21H (функции 39H, 3AH, 3BH, 47H).

1. *Создание каталога*.

Вход: AH = 39H, DS:DX – адрес строки с именем каталога в формате ”диск:\имя каталога”,0

Выход: если CF установлен (ошибка), то в AX – код ошибки.

2. *Удаление каталога*.

Вход: AH = 3AH, DS:DX – адрес строки с именем каталога в формате ”диск:\имя каталога”,0

Выход: если CF установлен (ошибка), то в AX – код ошибки.

3. *Получение имени текущего каталога.*

Вход: AH = 47H, DS:SI – адрес локального буфера для результирующего пути (64 байта); DL – номер диска (0 – текущий, 1 – A и т.д.)

Путь возвращается в формате ”путь\каталог”,0. Не подставляется впереди буква диска, а сзади не подставляется символ «\».

Выход: если CF установлен (ошибка), то в AX – код ошибки.

4. *Изменение текущего каталога.*

Вход: AH = 3BH, DS:DX – адрес строки с именем нового каталога в формате ”диск:\имя каталога”,0

Выход: если CF установлен (ошибка), то в AX – код ошибки.

ВАРИАНТ 3

Составить программу удаления файлов в соответствии с шаблоном (например ∗.obj) с подтверждением и без. Использовать следующие функции прерывания 21h:

1. *Удаление файла*.

Вход: AH = 41H, DS:DX – имя файла для удаления в формате ”диск:\путь\имя файла”,0

Выход: если CF установлен (ошибка), то в AX – код ошибки.

2. *Установить адрес DTA* – системной области данных с информацией о файле (44 байта).

Вход: AH = 1AH, DS:DX – адрес для DTA. Выход: нет.

3. *Найти первый файл в соответствии с шаблоном*.

Вход: AH = 4EH, DS:DX – адрес строки с именем файла в формате ”диск:\путь\имя файла”,0 (допускаются символы ? и *), CX – атрибут файла для поиска (для обычного файла – 20H).

Выход: если CF = 1, то в AX – код ошибки, в DTA – информация о файле: DTA+15H (1 байт) – атрибут файла, DTA+16H (2 байта) – время создания/модификации файла, DTA+18H (2 байта) – дата создания/модификации файла, DTA+1AH (4 байта) – размер файла в байтах, DTA+1EH (13 байт) – имя файла в виде ”имя_файла.расширение”,0

4. *Найти следующий файл в соответствии с шаблоном*.

Вход: AH = 4FH, DS:DX – адрес данных, возвращенных предыдущей (1EH) операцией поиска по шаблону.

Выход: если CF = 1, то в AX – код ошибки, в DTA – информация о файле.

ВАРИАНТ 4

Составить программу создания копии некоторого символьного файла. Использовать следующие функции прерывания 21h:

1. *Открытие файла с помощью номера (описателя) файла*.

Вход: AH = 3DH, AL – режим открытия (0 – Read Only (только чтение), 1 – Write Only (только запись), 2 – Read/Write (чтение/запись)), DS:DX – имя файла в формате ”диск:\путь\имя файла”,0

Выход: если CF=1, то в AX – код ошибки, иначе в AX – описатель файла.

2. *Закрытие файла.*

Вход: AH = 3EH, BX – описатель файла.

Выход: если CF = 1, то в AX – код ошибки.

3. *Создание файла.*

Вход: AH = 3CH, CX = 20H (архивный файл), DS:DX – имя файла в формате ”диск:\путь\имя файла”,0

Выход: если CF = 1, то в AX – код ошибки, иначе в AX – описатель файла.

4. *Чтение из файла.*

Вход: AH = 3FH, BX – описатель файла, CX – количество байт для чтения, DS:DX – адрес промежуточного буфера.

Выход: если CF = 1, то в AX – код ошибки, иначе в AX – количество действительно прочитанных байт.

5. *Запись в файл.*

Вход: AH = 40H, BX – описатель файла, CX – количество записываемых байт, DS:DX – адрес буфера.

Выход: если CF = 1, то в AX – код ошибки, иначе AX – количество действительно записанных байт.

ВАРИАНТ 5

Составить программу, выдающую следующую системную информацию: версию ОС, информацию о диске. Используйте следующие функции прерывания 21h:

1. *Дать версию ОС*.

Вход: AH = 30H

Выход: AL – старшая часть версии; AH – меньшая часть версии.

2. *Дать информацию о диске*.

Вход: AH=36H, DL = код устройства (0 – по умолчанию, 1 – A, 2 – B и т.д.)

Выход: AX – кол-во секторов на кластер (или 0FFFFH, если ошибка).
CX – кол-во байт/сектор;
BX – кол-во доступных кластеров;
DX – всего кластеров на диске.

Примечание: AX∗CX∗BX – размер свободного пространства на диске; AX∗CX∗DX – размер дискового пространства.

ВАРИАНТ 6

Составить программу установки размера/формы курсора и подавления курсора методом меню. Используйте функцию 01H прерывания 10h.

Установка размера/формы курсора.

Вход: AH = 01H; CH – начальная строка (0 – 1FH; 20H – подавить курсор); CL – конечная строка (0 – 1FH).

Выход: нет.

ВАРИАНТ 7

Составить программу запоминания и изменения позиции курсора в текстовом режиме. Используйте следующие функции прерывания 10h.

1. *Чтение позиции курсора.*

Вход: AH = 03H, BH – номер видеостраницы (0 – текущая страница).

Выход: DH, DL – текущие строка, столбец курсора; CH, CL – текущие начальные и конечные линии курсора (0 – 1FH; 20H – подавить курсор).

2. *Изменение позиции курсора.*

Вход: AH = 02H, DH, DL – строка, столбец (начиная с 0,0) (установка на 25 строку делает курсор невидимым); BH – номер видеостраницы.

Выход: нет.

3. *Вывод строки символов на экран*.

Вход: AH = 13H; ES:BP – выводимая строка; CX – длина строки; DH, DL – строка, колонка начала вывода; BL – атрибут; BH – номер видеостраницы (по умолчанию – 0); AL – код подфункции:

0 = использовать атрибут в BL; не трогать курсор

1 = использовать атрибут в BL; курсор – в конец строки

2 = формат строки: символ, атрибут, символ, атрибут…; не трогать курсор

3 = формат строки: символ, атрибут, символ, атрибут…; передвинуть курсор

Выход: нет.

Примечание: использовать подфункцию с кодом 0.

ВАРИАНТ 8

Составить программу вертикального скроллинга содержимого окна экрана. Используйте следующие функции прерывания 10h.

Скроллинг вверх (ah = 06h), вниз (ah = 07h).

Вход: AL – кол-во строк, скроллируемых в окне (AL=0 – очистка окна); BH – видео-атрибут, используемый для вдвигаемых строк; CH, CL – строка, столбец левого верхнего угла окна скроллирования; DH, DL – строка, столбец правого нижнего угла окна скроллирования.

Выход: нет.

ВАРИАНТ 9

Составить программу ввода (без отображения) и идентификации пароля (строки символов). Используйте функцию 08h прерывания 21h.

Ввод с клавиатуры без отображения.

Вход: AH = 08H. Выход: AL = код ASCII символа.

ВАРИАНТ 10

Составить процедуры сохранения и восстановления прямоугольной области экрана. Используйте функции 02H, 08H, 09H прерывания 10h.

1. *Установка позиции курсора*.

Вход: AH = 02H; BH – номер видеостраницы; DH, DL – строка и колонка соответственно, считая от 0.

Выход: нет.

2. *Чтение символа и атрибута в текущей позиции курсора.*

Вход: AH = 08H; BH – номер видеостраницы.

Выход: AL – прочитанный символ; AH – прочитанный видео-атрибут.

3. *Запись символа и атрибута в текущей позиции курсора*.

Вход: AH = 09H; BH – номер видеостраницы; AL – записываемый символ; BL – записываемый видео-атрибут; CX – количество записываемых экземпляров символа.

Выход: нет.

ВАРИАНТ 11

Составить процедуру ввода строки символов с клавиатуры и вывода строки символов на экран. Используйте прерывание 21h (функция 0AH) и прерывание 10h (функция 13H).

1. *Ввод строки в буфер*.

Вход: AH – 0AH; DS:DX – адрес входного буфера.

При входе буфер должен быть оформлен: max,?,?,?,?..., где max – максимально допустимая длина ввода (от 1 до 254).

Выход: буфер содержит ввод, заканчивающийся символом с кодом 0DH в виде: max, length, TEXT, 0DH, где length – действительная длина введенных данных (без символа 0DH).

2. *Вывод строки символов на экран.*

Вход: AH = 13H; ES:BP – выводимая строка; CX – длина строки; DH, DL – строка, колонка начала вывода; BL – атрибут; BH – номер видеостраницы (по умолчанию – 0); AL – код подфункции:

0 = использовать атрибут в BL; не трогать курсор

1 = использовать атрибут в BL; курсор – в конец строки

2 = формат строки: символ, атрибут, символ, атрибут…; не трогать курсор

3 = формат строки: символ, атрибут, символ, атрибут…; передвинуть курсор

Выход: нет.

ВАРИАНТ 12

Составить программу выбора и установки текущего диска методом меню. Используйте функции 19h, 0eh прерывания 21h.

1. *Дать текущий диск ДОС.*

Вход: AH = 19h; Выход: AL – номер текущего диска (0 = A, 1 = B и т.д.).

2. *Установить текущий диск*.

Вход: AH = 0Eh; DL – номер диска, ставшего текущим (0 = A, 1 = B и т.д.).

Выход: AL – общее число дисков в системе.

ВАРИАНТ 13

Составить программу вывода на экран имен всех файлов текущего каталога. Использовать следующие функции прерывания 21h:

1. *Установить адрес DTA* – системной области данных с информацией о файле (44 байта).

Вход: AH = 1AH, DS:DX – адрес для DTA. Выход: нет.

2. *Найти первый файл в соответствии с шаблоном*.

Вход: AH = 4EH, DS:DX – адрес строки с именем файла в формате ”диск:\путь\имя файла”,0 (допускаются символы ? и *), CX – атрибут файла для поиска (для обычного файла – 20H).

Выход: если CF = 1, то в AX – код ошибки, в DTA – информация о файле: DTA+15H (1 байт) – атрибут файла, DTA+16H (2 байта) – время создания/модификации файла, DTA+18H (2 байта) – дата создания/модификации файла, DTA+1AH (4 байта) – размер файла в байтах, DTA+1EH (13 байт) – имя файла в виде ”имя_файла.расширение”,0

3. *Найти следующий файл в соответствии с шаблоном*.

Вход: AH = 4FH, DS:DX – адрес данных, возвращенных предыдущей (1EH) операцией поиска по шаблону.

Выход: если CF = 1, то в AX – код ошибки, в DTA – информация о файле.

ВАРИАНТ 14

Составить процедуру копирования содержимого экрана в текстовом режиме в файл. Используйте для выполнения задания прерывания 10h и 21h. Соответствующая информация о функциях этих прерываний приведена в вариантах 4 и 10.

ВАРИАНТ 15

Составить программу получения и изменения текущего времени. Используйте прерывание 21h (функции 2CH, 2DH).

1. *Получение текущего времени*.

Вход: AH = 2CH;

Выход: CH – часы, CL – минуты, DH – секунды, DL – сотые доли секунд.

2. *Установка текущего времени*.

Вход: AH = 2DH, CH – часы, CL – минуты, DH – секунды, DL – сотые доли сек.

Выход: AL = 00H – если нет ошибок; FFH – если некорректные данные.

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА № 2
ТЕМА: ПРОГРАММИРОВАНИЕ TSR-ПРОГРАММ

Цель: знакомство с основами организации и проблемами при написании TSR-программ в среде MS DOS

1. Методические указания к выполнению работы

1.1 TSR-программы. Общие положения

TSR-программы (англ. Terminate and Stay Resident) отличаются от обычных программ тем, что после своего завершения остаются (в основном частично, то есть отдельными модулями или процедурами) в памяти компьютера, а операционная система защищает «занятую» ими память от повторного использования. Использование TSR-программ позволяет расширить возможности MS-DOS, являющейся по своей сути однозадачной системой. Неприспособленность MS-DOS к многопрограммной работе обуславливает особые требования к TSR, которые подробнее рассматриваются далее.

Операционная система имеет две возможности для резидентного завершения программ:

1) прерывание 27h (только для COM-файлов);

2) функция 31h прерывания 21h (для EXE-файлов и COM-файлов).

Небольшие различия между ними обусловлены способом, который использует операционная система для определения размера блока памяти, объявляемого резидентным. Исполняя прерывание 27h, операционная система принимает за начало блока значение в регистре CS, а длина блока памяти в байтах задается в регистре DX. При выполнении функции 31h прерывания 21h значение DX задает длину блока в параграфах, а за начало блока принимается PID – параграф, по которому в памяти располагается префикс программного сегмента (PSP – Program Segment Prefix) исполняемой в данный момент программы.

Структурно TSR-программа состоит из двух функционально-различных секций: *резидентной и инициализирующей.*

Инициализирующая часть выполняется только один раз – при запуске программы на выполнение. Резидентная часть TSR состоит из обработчиков прерывания (ISR – Interrupt Service Routine), составленных с применением специальных правил. *Обработчики получают управление при возникновении каких-либо внешних условий, генерирующих аппаратное прерывание, или при возникновении программного прерывания*. Точки входа в ISR записываются в таблицу векторов прерывания (в этом случае говорят, что прерывание «перехвачено»).

ISR – процедура должна быть абсолютно «незаметной» для прерываемых программ. Для этого она должна удовлетворять следующим требованиям:

1) ISR должна начинаться секцией кода, сохраняющей изменяемые в процессе работы регистры;

2) ISR должна заканчиваться процедурой восстановления измененных регистров;

3) возврат из ISR должен осуществляться командой IRET.

Функции инициализирующей части программы:

1) перехват прерывания;

2) резидентное завершение TSR-программы;

3) предотвращение повторной установки TSR-программы;

4) освобождение памяти, занятой TSR-программой.

Существует два основных механизма перехвата прерывания:

1) каскадное включение;

2) переопределение «старого» обработчика.

Каскадное включение – это такая установка в систему «нового» обработчика прерывания, при которой последний получает управление, а затем вызывает «старый» обработчик прерывания.

Переопределение – это способ подключения ISR, при котором «новый» обработчик полностью подменяет собой «старую» ISR.

Существует несколько возможностей для записи в таблицу векторов прерываний «новой» ISR:

1) непосредственный доступ;

2) использование функции 25h прерывания 21h.

Непосредственный доступ к таблице векторов прерываний – самый быстрый способ. Однако манипуляции с векторами требуют особой осторожности. Необходимо установить флаг IF в 0, блокируя аппаратные прерывания, а после завершения работы с таблицей прерываний вновь разрешить аппаратные прерывания, установив IF в 1.

1.2 Предотвращение повторной загрузки

Предотвращение повторной загрузки может осуществляться:

1) сканированием цепочки MCB (Memory Control Block) – блоков и определение того, имеется ли в памяти блок, «хозяином» которого является TSR с известным именем;

2) использованием специальных сигнатур, помещенных в ISR;

3) использованием «эха» в цепочке ISR.

Первый из методов позволяет обнаружить наличие в памяти резидентной части TSR, но не отвечает на вопрос, работоспособна ли она.

Идея второго способа заключается в следующем. Из таблицы считывается вектор прерывания, который используется ISR. По адресу данного вектора проверяется наличие определенной сигнатуры. При наличии таковой делается вывод, что TSR в памяти присутствует. Однако этот способ не сработает, если прерывание было повторно перехвачено.

Третий метод избавлен от этого недостатка. Одно из прерываний перехватывается каскадным методом. При вызове данного прерывания с определенными уникальными значениями регистров (ключом), ISR возвращает в одном или нескольких регистрах условленные уникальные значения (эхо). Правильно выбрав прерывание, ключ и эхо, можно с высокой надежностью определить наличие в памяти рабочей копии TSR.

1.3 «Горячая» клавиша

Для активизации TSR часто используется некоторая комбинация клавиш («горячая» клавиша»). Для того, чтобы TSR реагировала на «горячую» клавишу, как правило, собственные ISR этого прерывания включаются по каскадной схеме. Во-первых, нельзя отключить стандартный BIOS-обработчик прерывания 9, который выполняет множество разных функций в интересах всей системы. Во-вторых, использование каскадной схемы не «топит» другие TSR, перехватившие это прерывание ранее.

Общая схема построения ISR прерывания от клавиатуры такова. ISR анализирует наступление момента активизации, задаваемого состоянием Shift- и триггерных (NumLock, CapsLock, ScrollLock) клавиш, возможно, в сочетании с обычной клавишей. Определение состояния Shift- и триггерных клавиш можно выполнять непосредственным доступом в память или с помощью прерывания 16h. Признаком нажатия нужной клавиши будет появление в буфере клавиатуры нужного двухбайтового BIOS-кода клавиши. Из этих двух байт лучше ориентироваться на старший, где помещается скан-код клавиши. Еще более надежным будет чтение скан-кода не из буфера клавиатуры, а непосредственно из порта с шестнадцатеричным значением 60. Например, командой IN AL, 60H. В этом случае после обнаружения кода клавиши его необходимо удалить из буфера.

1.4 Проблема реентерабельности

Реентерабельная программа – это программа, которая разрешает, в силу особенностей своего построения, начинать ее выполнение несколько раз, не дожидаясь завершения выполнения программы, начатого ранее. Она не изменяет ни одной константы или переменной, которые могут повлиять на повторное выполнение программы. Большинство программ, образующих в совокупности ядро операционной системы, не являются реентерабельными. Отсюда вытекает требование к ISR активизировать TSR только тогда, когда операционная система позволяет повторное вхождение.

Для этого существуют две основные возможности:

1) использование прерывания 28h;

2) использование флага повторного вхождения в MS-DOS.

В первом случае программа перехватывает прерывание 28h, которое получает управление в те моменты времени, когда MS-DOS выполняет цикл ожидания ввода с клавиатуры и перед исполнением функций с номерами от

0h до 0Ch. Ограничение, накладываемое на TSR в этом случае состоит в запрете на использование функций MS-DOS с номерами от 0h до 0Ch.

Другой способ предотвращения повторного вхождения в MS-DOS основан на анализе состояния флага реентерабельности MS-DOS, адрес которого возвращается недокументированной функцией 34h MS-DOS. Его значение равно 1, когда MS-DOS находится в нереентерабельной части своего кода и наоборот.

1.5 Проблема переключения PSP

Идентификатором активной программы для MS-DOS служит PID – сегмент, по которому в памяти располагается префикс программного сегмента – PSP. В PSP сосредоточена вся информация, необходимая MS-DOS для управления памятью и программами (в частности, в PSP содержится командная строка, исполненная при вызове программы). В некоторых случаях TSR не может использовать PSP текущей программы. В этом случае она должна выполнить переключение с помощью функций 62h и 50h прерывания 21h.

1.6 Справочная информация

PSP – это область памяти, размером 256 байт, расположенная непосредственно перед началом кода программы. Эта область содержит информацию, необходимую операционной системе для работы программы, а также обеспечивает место для файловых операций ввода/вывода (таблица 2.1).

Таблица 2.1 – Структура PSP

Смещение	Размер (в байтах)	Значение
00h	2	Номер функции DOS завершения программы
02h	2	Размер памяти в параграфах
04h	2	Резерв
06h	4	Длинный вызов функции диспетчера DOS
0Ah	4	Адрес завершения (IP, CS)
0Eh	4	Адрес выхода по Ctrl-Break (IP, CS)
12h	4	Адрес выхода по критической ошибке
16h	22	Резерв
2Ch	2	Номер параграфа строки среды
2Eh	46	Резерв
5Ch	16	FCB 1
6Ch	20	FCB 2
80h	128	Область DTA (по умолчанию – командная строка программы)

1.7 Прерывания и функции DOS

Функция 35h прерывания 21h – получить вектор прерывания.
Вход: AH = 35h; AL = номер прерывания.
Выход: ES:BX = адрес обработчика прерывания.

Функция 25h прерывания 21h – установить вектор прерывания.
Вход: AH = 25h; AL = номер прерывания; DS:DX = адрес обработчика прерывания.
Выход: нет

Прерывание 27h. Прерывает выполнение программы и оставляет резидентную часть в памяти.
Вход: DX = адрес первого байта за резидентным участком программы.
Выход: нет.

Функция 31h прерывания 21h прерывает выполнение программы и оставляет резидентную часть в памяти.
Вход: AH = 31h; AL = код возврата; DX = объем памяти, оставляемой резидентной, в параграфах.
Выход: нет.

Функция 62h: получить адрес PSP.
Вход: AH = 62h.
Выход: BX = сегментный адрес PSP выполняющейся программы.

1.8 Пример кода TSR-программы

Приведенный ниже код программы показывает, как осуществить перехват прерывания от клавиатуры (вектор номер 9).

```
codesg      segment  para  'code'
            assume   cs : codesg, ds : codesg, es : codesg, ss : codesg
            org    100h
begin:      jmp    main

; ———— процедура обработки прерывания ————————————
rezident    proc   far
            push   ds di es ax bx cx dx ; сохраняем регистры в стеке
            in     al,60h       ; читаем скан код клавиши из порта 60h
            cmp    al,16        ; это скан-код клавиши Q ?
            je     @my_job      ; да – уходим на свою обработку
            int    60h          ; вызываем оригинальный обработчик
            pop    dx cx bx ax es di ds ; восстанавливаем все регистры
            iret                ; возврат из обработчика прерывания
@my_job:    pop    dx cx bx ax es di ds ; восстанавливаем все регистры
            mov    al,20h       ; посылаем сигнал контроллеру о
            out    20h, al      ; завершении аппаратного прерывания
            iret                ; возврат из обработчика прерывания
finish      EQU    $            ; адрес первого байта за резидентной частью
rezident  endp
; ———— конец процедуры обработки прерывания ————————
```

```
;————————— инициализирующая часть ———————————————————
main   proc   near
; ——  1) получаем вектор прерывания номер 9 в регистры es:bx ———
       mov   ah,35h        ; номер функции получения вектора
       mov   al,09h        ; номер вектора
       int   21h           ; вызвать прерывание
; ——— 2) переписываем полученный вектор в 60-ый вектор (пустой) ——
       cli                 ; запрещаем аппаратные прерывания
       push  ds            ; сохраняем в стеке ds нашей программы
       mov   dx,bx         ; переписываем bx в dx
       mov   ax,es         ; переписываем es в ds
       mov   ds,ax         ; через регистр ax
       mov   ah,25h        ; номер функции установки вектора
       mov   al,60h; номер вектора, куда пишем
       int   21h           ; вызвать прерывание
       pop   ds            ; восстанавливаем из стека ds нашей программы
       sti                 ; разрешаем аппаратные прерывания
; –  3) записываем адрес нашей процедуры обработки в вектор номер 9 –
       lea   dx,rezident   ; загрузка в DS:DX адреса резидентной процедуры
       mov   ah,25h        ; функция установки  вектора
       mov   al,09h; номер вектора, куда пишем
       int   21h           ; вызвать прерывание
; — 4) прерываем программу и оставляем резидентную часть в памяти —
       lea   dx, finish    ; в dx номер первого байта за резидентной частью
       int   27h           ; вызвать прерывание
       ret
main  endp
codesg       ends
       end   begin
```

Клавиша Q обрабатывается нашей резидентной процедурой resident непосредственно, а все остальные клавиши – оригинальной процедурой обработки прерывания, вызываемой процедурой resident.

2. Контрольные вопросы по теме

1. Структура TSR – программы.
2. Что значит «перехватить» прерывание?
3. Как микропроцессор организует вызов прерывания?
4. Как проверить, находится ли уже TSR – программа в памяти?
5. Назначение PSP. Как проанализировать командную строку средствами Ассемблера?
6. Опишите алгоритм обработки «горячей» клавиши.
7. Каковы основные проблемы, возникающие при написании TSR – программ? Чем они вызваны?

3. Варианты индивидуальных заданий

Составить TSR – программу на языке Ассемблера в соответствии с номером варианта.

Примечания.

1. Программа не должна загружаться при повторном запуске.
2. Программа должна активизироваться при нажатии «горячей» клавиши.

ВАРИАНТ 1

Составить программу «часы», резидентно отсчитывающую время после запуска. Используйте прерывания 08H или 1CH и 10H.

ВАРИАНТ 2

Составить программу «будильник», выдающую на экран строку «Время истекло!» после запуска по истечении некоторого промежутка времени (в секундах). Время задержки задается в командной строке. Используйте прерывание 08H или 1CH и 10H.

ВАРИАНТ 3

Составить программу перемещения первой строки экрана в последнюю строку экрана по истечении некоторого промежутка времени, заданного в командной строке. Используйте прерывание 08H или 1CH и 10H..

ВАРИАНТ 4

Составить программу вывода произвольного прямоугольного окна экрана, координаты которого задаются в командной строке, в другую часть экрана с координатами левого верхнего левого угла, задаваемыми в командной строке. Используйте прерывание 10H.

ВАРИАНТ 5

Составить программу очистки произвольного прямоугольного окна экрана, координаты которого задаются в командной строке. Используйте прерывание 10H.

ВАРИАНТ 6

Составить программу «динамического» форматирования текста на экране, т.е. курсор автоматически переносится на следующую строку при достижении определенной позиции, вводимой в командной строке. Используйте прерывание 10H.

ВАРИАНТ 7

Составить программу «динамического» перезапуска компьютера по истечении некоторого промежутка времени. Используйте прерывание 08H и 19H.

ВАРИАНТ 8

Составить программу перевода прописных латинских букв в строчные буквы и строчных букв в прописные буквы в строке, на которой установлен курсор. Используйте прерывание 10Н.

ВАРИАНТ 9

Составить программу заполнения пробелом начальной или конечной части строки, указанной курсором. Вариант заполнения вводится в командной строке. Один конец части строки указывается текущим положением курсора. Используйте прерывание 10Н.

ВАРИАНТ 10

Составить программу выдачи краткого комментария, если курсор находится в начале или конце ключевого слова. Порядок расположения слова (слева или справа от курсора) определяется командной строкой. Используйте прерывание 10Н.

ВАРИАНТ 11

Составить программу перемещения курсора из левого верхнего угла экрана в правый нижний угол по истечении промежутка времени, указанного в командной строке. Используйте прерывания 08Н или 1СН и 10Н.

ВАРИАНТ 12

Составить программу замены на пробелы символов псевдографики на экране при нажатии горячей клавиши. Используйте прерывание 10Н.

ВАРИАНТ 13

Составить программу выдачи скан-кода символа, указанного курсором из множества символов {0…9}. Используйте прерывание 10Н.

ВАРИАНТ 14

Составить программу – калькулятор, выполняющую операции сложения и вычитания над целыми цифрами. Калькулятор срабатывает при нажатии на «горячую» клавишу и вычисляет выражение, заданное в командной строке. Используйте прерывание 10Н.

ВАРИАНТ 15

Составить программу увеличения размера курсора или его подавления по истечении промежутка времени, заданного в командной строке. Режим увеличения или подавления также задается в командной строке. Используйте прерывания 08Н и 10Н.

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА № 3
ТЕМА: ТАЙМЕР И ЗВУК

Цель: знакомство с методами создания звуковых эффектов с помощью встроенного динамика

1. Методические указания к выполнению работы

Микросхема таймера 8253 работает независимо от процессора. Процессор программирует микросхему, а затем переходит к другим делам. Микросхема 8253 имеет три независимых идентичных канала, которые могут программироваться. Каждый из трех каналов микросхемы таймера 8253 состоит их трех регистров.

Доступ к каждой группе из трех регистров осуществляется через один порт. Номера портов от 40h до 42h соответствуют каналам 0, 1 и 2. Порт связан с 8-битовым регистром ввода-вывода, который посылает и принимает данные для этого канала. Когда канал запрограммирован, через этот порт посылается 2-байтовое значение. Младший байт посылается первым. Это число передается в 16-битовый регистр задвижки (latch register), который хранит это число, и из которого копия помещается в 16-битовый регистр счетчика. В регистре счетчика число уменьшается на 1 каждый раз, когда импульс от системных часов пропускается через канал. Когда значение этого регистра достигает нуля, канал выдает выходной сигнал, и затем новая копия содержимого регистра задвижки передвигается в регистр счетчика, после чего процесс повторяется. Все три канала всегда активны, процессор не включает и не выключает их. Текущее значение любого из регистров счетчика может быть прочитано в любой момент.

Каждый канал имеет две входные и одну выходную линии. Выходная линия выводит импульсы, возникающие в результате подсчета. Назначение этих сигналов варьируется в зависимости от персонального компьютера.

Канал 0 используется системными часами времени суток. Он устанавливается BIOS при старте таким образом, что выдает импульсы приблизительно 18.2 раза в секунду. 4-байтовый счетчик этих импульсов хранится в памяти по адресу 0040:006C (младший байт хранится первым). Каждый импульс инициирует прерывание таймера, и именно это прерывание увеличивает значение счетчика. Это прерывание является аппаратным прерыванием. Выходная линия используется также для синхронизации некоторых дисковых операций.

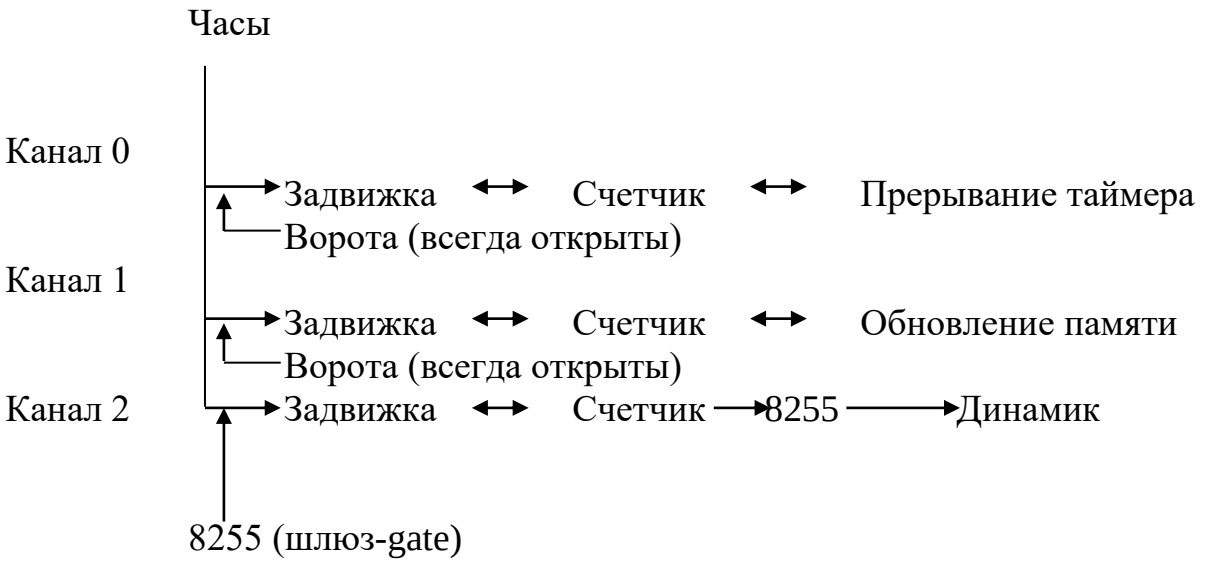
Канал 1 управляет обновлением памяти, поэтому его лучше не трогать. Выходная линия этого канала связана с микросхемой прямого доступа к памяти, и ее импульс заставляет микросхему DMA (Direct Memory Access – Прямой Доступ в Память) обновить всю память.

Канал 2 микросхемы прямо связан с динамиком компьютера. Когда этот канал программируется в режиме 3, то он производит простые

прямоугольные волны заданной частоты. Из-за простоты динамика он сглаживает края прямоугольной волны, получая более приятную для слуха синусоидальную волну. К сожалению, микросхема 8253 не может менять амплитуду волны, поэтому нельзя изменить громкость звука динамика.

Простые звуки могут генерироваться одновременно с другими программными операциями, а более сложные звуковые эффекты могут быть достигнуты за счет использования процессора. Надо отметить, что динамик не будет генерировать звук до определенных установок микросхемы интерфейса с периферией 8255.

Две входные линии для каждого канала состоят из линии часов, передающей сигнал от микросхемы системных часов и линии, называемой воротами (gate), которая включает и выключает сигнал от часов. Ворота всегда открыты для сигналов часов по каналам 0 и 1. Однако они могут быть закрытыми для канала 2, что позволяет производить некоторые специальные манипуляции со звуком. *Ворота закрываются установкой младшего бита порта с адресом 61h*, который является регистром микросхемы 8255. *Сброс этого бита в ноль вновь открывает ворота.* Отметим тот факт, что, как и выход канала 2, бит 1 порта 61h связан с динамиком и также может использоваться для генерации звука.



Микросхема таймера может применяться непосредственно для временных операций, но это редко бывает удобным. Ввод с часов производится 1.19318 раз в секунду. Поскольку максимальное число, которое может храниться в 16 битах, равно 65535, и оно делится на частоту импульсов от часов, равную 18.2, то максимально возможный интервал между импульсами равен приблизительно 1/12 с. Поэтому в большинстве временных операций используется счетчик времени суток BIOS.

Микросхема таймера 8253 предоставляет разработчикам 6 режимов работы для каждого канала. *Программисты обычно ограничиваются третьим режимом как для канала 0 при синхронизации, так и для канала 2 при генерации звука.* В этом режиме, как только регистр задвижки

получает число, он немедленно загружает копию в регистр счетчика. Когда значение в счетчике достигает нуля, регистр задвижки мгновенно перезагружает счетчик и т.д. В течение первой половины отсчета выходная линия включена, в течение второй – выключена. В результате получаются прямоугольные волны, вполне пригодные для генерации звука.

Восьмибитовый командный регистр управляет способом загрузки чисел в канал. Адрес порта для этого регистра равен 43h. Командному регистру передается байт, который сообщает, какой канал программировать, в каком режиме, тип операции и будет ли число в двоичной или двоично-десятичной форме (BCD). Значения битов этого регистра показаны в таблице 3.1.

Таблица 3.1 – Значение битов командного регистра (43H)

Биты	Значение
0	если 0, двоичные данные, иначе двоично-десятичные данные
3–1	номер режима, 0–5 (000–101)
5–4	тип операции: 00 = передать значение счетчика в задвижку 01 = читать/писать только старший байт 10 = читать/писать только младший байт 11 = читать/писать сначала младший байт, а затем старший
7–6	номер программируемого канала 0–2 (00–10)

Для программирования микросхемы 8253 надо выполнить три шага:

1) послать в командный регистр (43h) байт, представляющий цепочку битов, которые выбирают канал, статус чтения/записи, режим операции и форму представления чисел. Например, цепочка 10 11 011 0 означает, что программируется второй канал для генерации звука, записывается сначала младший байт, а потом старший, используется третий режим работы канала, передаются двоичные данные;

2) для канала 2 надо разрешить сигнал от часов, установив в 1 бит 0 порта с адресом 61h. Если бит 1 этого регистра установить в 1, то канал 2 будет управлять динамиком;

3) установить значение счетчика от 0 до 65535, поместить его в AX и послать сначала младший, а затем и старший байт в регистр ввода-вывода канала (40h–42h).

После того как третий шаг завершен, запрограммированный канал немедленно начинает функционировать по новой программе.

Каналы микросхемы 8253 работают постоянно. Поэтому программы должны всегда восстанавливать начальные установки регистров 8253 перед завершением. В частности, если при завершении программы генерируется звук, то он будет продолжаться даже после того, как операционная система получит управление и загрузит другую программу.

В таблице 3.2. приведена частота нот хроматической гаммы.

Таблица 3.2. – Частота нот хроматической гаммы

До	До#	Ре	Ре#	Ми	Фа	Фа#	Соль	Соль#	Ля	Ля#	Си	Октава
1	2	3	4	5	6	7	8	9	10	11	12	Контроктава
33	35	37	39	41	44	46	49	52	55	58	62	
13	14	15	16	17	18	19	20	21	22	23	24	Большая
65	69	73	78	82	87	92	98	104	110	117	123	
25	26	27	28	29	30	31	32	33	34	35	36	Малая
131	139	147	156	165	175	185	196	208	220	233	247	
37	38	39	40	41	42	43	44	45	46	47	48	Первая
262	277	294	311	330	349	370	392	415	440	466	494	
49	50	51	52	53	54	55	56	57	58	59	60	Вторая
523	554	587	622	659	698	740	784	831	880	932	988	
61	62	63	64	65	66	67	68	69	70	71	72	Третья
1047	1109	1175	1245	1319	1397	1480	1568	1661	1760	1865	1976	
73	74	75	76	77	78	79	80	81	82	83	84	Четвертая
2093	2217	2349	2489	2637	2794	2960	3136	3322	3520	3729	3951	

Приведенный ниже фрагмент кода программы заставляет динамик генерировать звук с частотой 1047 (нота «До» третьей октавы).

```
mov     dx,1047             ; загрузить в dx частоту ноты «До»
; 1 – послать управляющий код в порт 43h
mov     al,10110110b
out     43h,al
; 2 – разрешить динамик и таймер
in      al,61h              ; прочитать байт из порта 61h
or      al,00000011b        ; установить 0-ой и 1-ый бит в 1
out     61h,al              ; записать байт в порт 61h
; 3 – послать частоту ноты в порт 42h
mov     al,dl               ; послать младший байт частоты
out     42h,al              ; в порт 42h
mov     al,dh               ; послать старший байт частоты
out     42h,al              ; в порт 42h
```

Приведенный ниже фрагмент кода программы выключает динамик.

```
in      al,61h              ; прочитать байт из порта 61h
and     al,11111100b        ; установить 0-ой и 1-ый бит в 0
out     61h,al              ; записать байт в порт 61h
```

Альтернативные способы генерации звука приведены ниже.

1. Генерация звука с помощью адаптера интерфейса с периферией 8255 состоит во включении и выключении с желаемой частотой бита 1 порта 61h. Бит 0 порта 61h управляет воротами канала 2 микросхемы таймера. Поэтому этот бит должен быть сброшен для отсоединения от канала таймера. Аппаратные прерывания должны быть запрещены.

2. Генерация простого гудка производится подачей символа BELL (ASCII-код 7) на стандартное устройство вывода.

2. Контрольные вопросы по теме

1. Структура микросхемы таймера 8253.
2. Каким образом в управлении динамиком принимает участие микросхема интерфейса с периферией?
3. Перечислите каналы микросхемы таймера и их назначение.
4. Приведите структуру командного регистра микросхемы таймера.
5. Укажите особенности режима номер 3 работы каналов таймера.
6. Перечислите способы получения звука на персональной ЭВМ.
7. Приведите алгоритм программирования микросхемы таймера.
8. Почему на персональной ЭВМ типа IBM PC нет возможности регулирования громкости звука, издаваемого динамиком?

3. Варианты индивидуальных заданий

ВАРИАНТ 1

Написать программу «пианино», генерирующую ноты первой октавы при нажатии различных клавиш клавиатуры. Генерацию звука производить путём программирования таймера.

ВАРИАНТ 2

Написать программу проигрывания мелодии.

Таблица номеров частот и длительностей нот: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22.

Таблица частот: 1709, 1809, 2031, 2280, 1521, 1709, 1809, 2031, 2280, 1521, 1709, 1353, 1709, 1809,1521,1809, 2031, 1809, 1709, 2031, 2280, 1139.

Таблица задержек: 4, 4 ,4, 4, 5, 4, 4, 4, 4, 5, 4, 5, 4, 4, 5, 4, 4, 4, 4, 4, 5, 5.

Генерацию звука производить путём программирования таймера.

ВАРИАНТ 3

Написать программу получения плавного перехода тонов – «взлетающий самолет». Генерацию звука производить путём программирования таймера.

ВАРИАНТ 4

Написать программу «пианино», генерирующую ноты второй октавы при нажатии различных клавиш клавиатуры. Генерацию звука производить путём управления динамиком непосредственно процессором (использование таймера запрещено).

ВАРИАНТ 5

Написать программу проигрывания мелодии.

Таблица номеров частот и длительностей нот: 1, 9, 5, 9, 4, 9, 2, 5, 4, 1, 9, 5, 11, 1, 11, 8, 11, 10, 5, 6, 7, 12, 8, 5, 2, 1, 5, 4, 10, 11, 10, 5, 4, 9, 5, 4, 1.

Таблица частот: 2280, 2031, 1809, 1709, 1521, 1353, 1207, 1139, 1920, 1437, 1280, 1003, 0, 890, 1615, 2155, 936.

Таблица задержек: 6, 6, 6, 6, 12, 6, 6, 12, 12, 18, 6, 6, 6, 0, 6, 12, 6, 6, 18, 18, 18, 6, 6, 12, 12, 6, 6, 6, 18, 6, 6, 12, 6, 6, 12, 12, 18.

Генерацию звука производить путём программирования таймера.

ВАРИАНТ 6

Написать программу получения плавного перехода тонов – «разгоняющийся автомобиль». Генерацию звука производить путём управления динамиком непосредственно процессором (использование таймера запрещено).

ВАРИАНТ 7

Написать программу «пианино», генерирующую ноты третьей октавы при нажатии различных клавиш клавиатуры. Генерацию звука производить путём программирования таймера.

ВАРИАНТ 8

Написать программу проигрывания мелодии.

Таблица номеров частот и длительностей нот: 6,1,6,2,4,6,8,11,13,6,5,4,3,5,4,13,6,1,6,2,4,6,14,12,13,12,8,12,8,11,6,13,6,12,14, 12,8,1,8,11,13,11,8,12,8,11,1,11,6,13,6,1,6,5,15,13,5,17,12, 6,13,6,1,6,11,6,5,4.

Таблица частот: 2280,2031,1809,1709,1521,1353,1207,1139,1920,1437, 1280,1003,0, 890,1615,2155,936.

Таблица задержек: 6,0,12,6,12,6,6,12,6,12,6,12,6,6,12,12, 6,0,12,6,12,6,6,12, 12,6,12,6,12,6,18,6,6,12,12,6,6,0,12,6,6,6,12,12,6,6,0,12,12,6,6,0,3,3,12,12,6,3, 3,12,12,6,0,12,12,6,6,12.

Генерацию звука производить путём программирования таймера.

ВАРИАНТ 9

Написать программу получения плавного перехода тонов – «пожарная сирена». Генерацию звука производить путём программирования таймера.

ВАРИАНТ 10

Написать программу «пианино», генерирующую ноты четвертой октавы при нажатии различных клавиш клавиатуры. Генерацию звука производить путём управления динамиком непосредственно процессором (использование таймера запрещено).

ВАРИАНТ 11

Написать программу проигрывания мелодии.

Таблица номеров частот и длительностей нот: 6, 5, 4, 3, 1, 1, 6, 5, 4, 3, 1, 1, 6, 2, 2, 6, 5, 1, 1, 5, 4, 5, 6, 4, 3, 3, 6, 2, 2, 6, 5, 1, 1, 5, 4, 5, 6, 4, 3, 3.

Таблица частот: 1140, 1015, 1705, 1520, 1355, 1205.

Таблица задержек: 8, 8, 8, 8, 4, 4, 8, 8, 8, 8, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 4, 4.

Генерацию звука производить путём программирования таймера.

ВАРИАНТ 12

Написать программу получения плавного перехода тонов – «падающий метеорит». Генерацию звука производить путём управления динамиком непосредственно процессором (использование таймера запрещено).

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА № 4
ТЕМА: ПЕРЕПРОГРАММИРОВАНИЕ КЛАВИАТУРЫ

Цель: знакомство с основами организации и управления стандартной клавишной клавиатурой

1. Методические указания к выполнению работы

Клавиатура содержит микропроцессор, который воспринимает каждое нажатие на клавишу и выдает скан-код в порт 60h микросхемы интерфейса с периферией, расположенной на системной плате. Скан-код, это однобайтовое число, младшее 7 бит которого представляют идентификационный номер нажатой клавиши. Старший бит кода говорит о том, была ли клавиша нажата (бит = 1) или отпущена (бит = 0).

Когда скан-код выдается в порт 60h, вызывается прерывание клавиатуры (INT 09h). Процессор прекращает свою деятельность и выполняет процедуру, анализирующую скан-код. При поступлении кода от клавиши сдвига или переключателя изменение статуса записывается в память. Во всех остальных случаях скан-код трансформируется в код символа при условии, что он подается при нажатии клавиши (в противном случае скан-код отбрасывается). Сначала процедура определяет установку клавиш сдвига и переключателей, чтобы правильно получить вводимый код. Затем введенный код помещается в буфер клавиатуры, который является областью памяти, способной запоминать вводимые символы, пока программа слишком занята, чтобы обработать их.

1.1 Организация буфера клавиатуры

Буфер содержит двухбайтовые кода для каждого нажатия на клавишу. Для однобайтовых кодов первый байт содержит код ASCII, а второй скан-код клавиши. Для расширенных кодов первый байт содержит ASCII 0, а второй – номер расширенного кода.

Буфер устроен, как циклическая очередь, которую называют также буфером FIFO. Он занимает непрерывную область адресов памяти. Однако определенной ячейки памяти, которая хранит «начало строки» в буфере, нет. Вместо этого два указателя хранят позиции головы и хвоста строки символов, находящейся в буфере. Новые нажатия клавиш сохраняются в позициях, следующих за хвостом (более старших адресах памяти), и, соответственно, обновляется указатель хвоста буфера. После того, как израсходовано все буферное пространство, новые символы продолжают вставляться, начиная с самого начала буферной области. Поэтому возможны ситуации, когда голова строки в буфере имеет больший адрес, чем хвост. Когда буфер заполнен, новые вводимые символы игнорируются, и прерывание клавиатуры выдает гудок через динамик.

В то время как указатель на голову установлен на первый введенный символ, указатель на хвост установлен на позицию за последним введенным

символом. Когда оба указателя равны, буфер пуст. 32 байта буфера начинаются с адреса 0040:001E. Размер буфера является фиксированным. Увеличить его можно только одним способом: завести свой программный буфер и обрабатывать все прерывания клавиатуры. Указатели на голову и хвост расположены по адресам 0040:001A и 0040:001C соответственно. Хотя под указатели отведено 2 байта, используется только младший байт. Значения указателей меняются от 30 до 60, что соответствует позициям в области данных BIOS. Для очистки буфера надо просто установить значение ячейки 0040:001A равным значению ячейки 0040:001C.

На рис. 4.1 показана конфигурация буфера клавиатуры.

0040 : 003C			F
0040 : 003A			F
0040 : 0038			U
0040 : 0036			B
0040 : 0034	R		
0040 : 0032	E		
0040 : 0030	F		
0040 : 002E	F		
0040 : 002C	U		
0040 : 002A	B		
0040 : 0028			
0040 : 0026			
0040 : 0024			
0040 : 0022			
0040 : 0020			R
0040 : 001E			E
0040 : 001C	0040 : 0036	← Указатель хвоста →	0040 : 0022
0040 : 001A	0040 : 002A	← Указатель головы →	0040 : 0036

Рис. 4.1 – Конфигурация буфера клавиатуры

1.2 Использование клавиш-переключателей

Три типа клавиш-переключателей заставляют другие клавиши клавиатуры генерировать различные коды. Как правило, такие комбинации генерируют расширенные коды. Но в двух случаях они дают коды ASCII: когда используется клавиша «Shift» с клавишами алфавитно-цифровых символов. Кроме того нажатие клавиш в комбинации от Ctrl-A до Ctrl-Z дает ASCII коды от 1 до 26. Все остальные комбинации позволяют получить расширенные коды.

Недопустимые комбинации клавиш не производят кода вообще. За исключением комбинаций с «Ctrl-Alt», одновременное нажатие нескольких переключателей приводит к тому, что только один из них становится эффективным, причем приоритет у «Alt», затем «Ctrl» и затем «Shift». Другие комбинации клавиш можно сделать допустимыми, лишь переписав прерывание клавиатуры.

Один из байтов-статусов переключателей клавиатуры находится по адресу 0040:0018. Он служит для отображения информации о клавишах-переключателях и имеет следующую структуру (рис. 4.2)

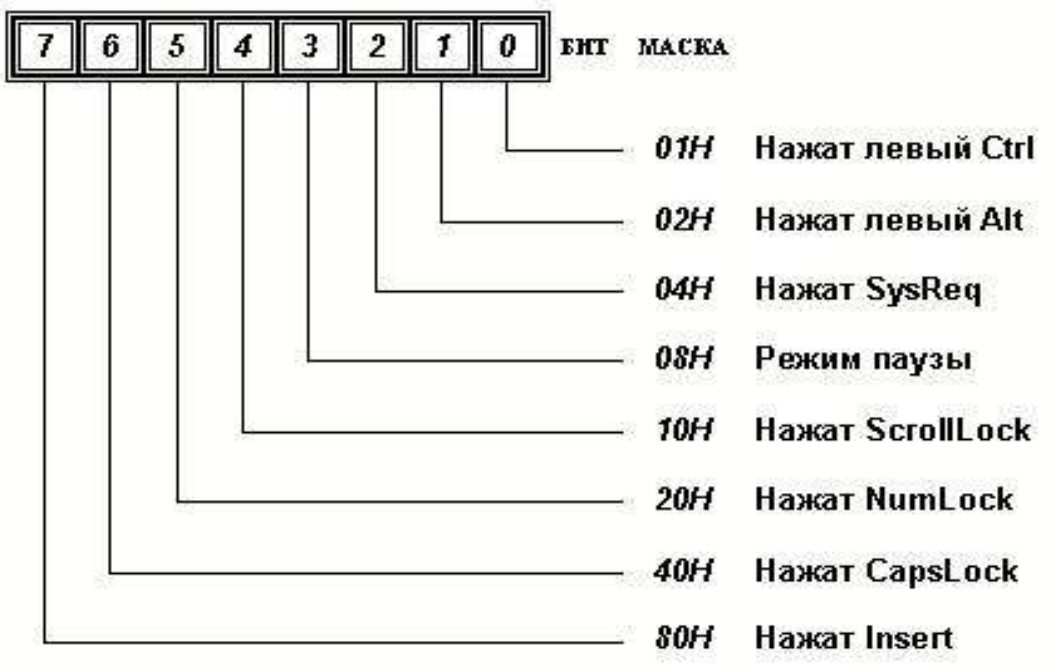


Рис. 4.2 – Байт-статус с адресом 0040:0018

Другой байт-статус переключателей клавиатуры находится по адресу 0040:0017 (рис. 4.3).

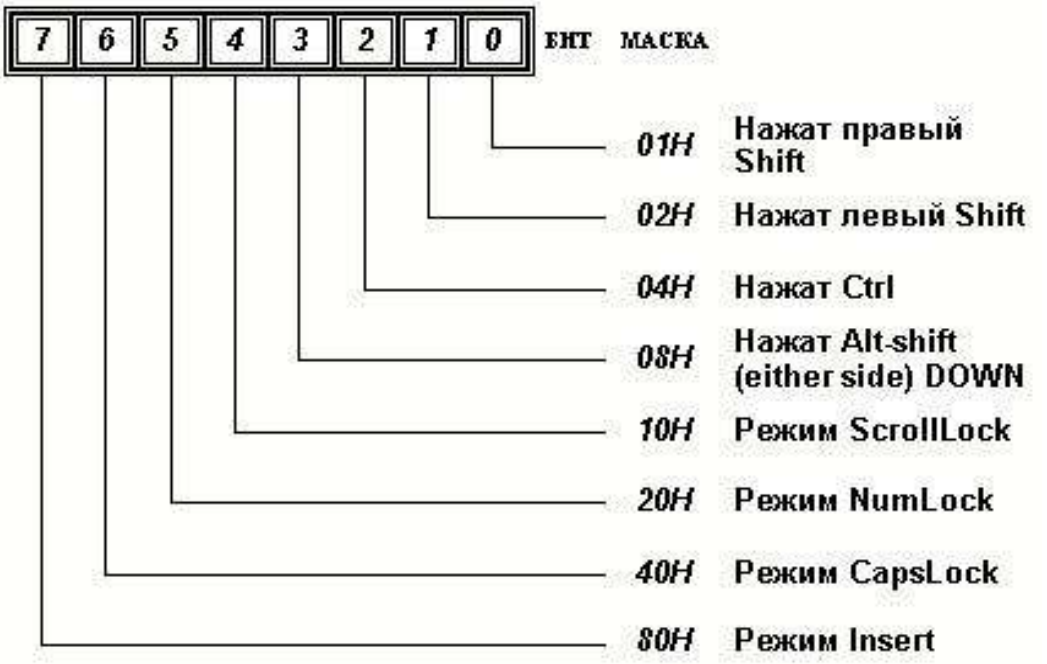


Рис. 4.3 – Байт-статус с адресом 0040:0017

Байты-статусы возвращает функция 02h прерывания 16h.

1.3 Перепрограммирование прерывания от клавиатуры

При перепрограммировании прерывании от клавиатуры необходимо выполнить четыре основных шага:

1) прочитать скан-код клавиши из порта с номером 60h:

```
in al,60h
```

2) послать клавиатуре подтверждающий сигнал, установив и сбросив седьмой бит порта с номером 61h:

```
in      al,61h
or      al,10000000b
out     61h,al
and     al,01111111b
out     61h,al
```

3) преобразовать скан-код в установку бита байта статуса клавиш-переключателей (например, если нажат правый Shift):

```
mov   ax,0040h
mov   es,ax
mov   bl,00000001b
or      es:[0017h],bl
```

или в номер кода, используя команду XLAT.

4) поместить код клавиши в хвост буфера клавиатуры.

2. Контрольные вопросы по теме

1. Общий алгоритм формирования кода ASCII.
2. Структура скан-кода для IBM PC.
3. Как организован буфер клавиатуры?
4. Как увеличить размер буфера клавиатуры?
5. Назначение байтов статуса клавиш-переключателей.

3. Варианты индивидуальных заданий

Составить резидентный «драйвер» клавиатуры, удовлетворяющий одному из нижеприведенных свойств.

ВАРИАНТ 1

«Гашение» экрана по истечению некоторого времени, в течение которого не было нажатий на клавиши.

Примечание: Гашение экрана VGA (прерывание 10h):
На входе: AH = 12h; BL = 36h – управление гашением экрана дисплея;
AL = 0 – включение экрана, 1 – гашение экрана.

ВАРИАНТ 2

Выдача «подсказки» по ключу /? во время запуска программы.

ВАРИАНТ 3

При переходе на кириллицу нажатие каждой клавиши подтверждается звуковым сигналом.

ВАРИАНТ 4

Деактивация драйвера по заданию ключа /r (/R).

ВАРИАНТ 5

Поддержка 2-х клавиш переключения режима: Ctrl – для переключения в режим кириллицы, Alt – для обратного перехода.

ВАРИАНТ 6

Проверка таких функций драйвера, как: загружен ли драйвер в память? Режим работы (лат/рус)?

ВАРИАНТ 7

Задание скан-кода клавиши переключения алфавита посредством командной строки.

ВАРИАНТ 8

Индикация режима работы драйвера строками Rus/Lat в верхнем левом углу экрана.

ВАРИАНТ 9

Поддержка 2-х клавиш переключения режима: Left Shift – для переключения в режим кириллицы, Right Shift – для обратного перехода.

ВАРИАНТ 10

Использование расширенного скан-кода для переключения алфавитов (например, Ctrl).

ВАРИАНТ 11

Поддержка символов псевдографики в национальном алфавите.

ВАРИАНТ 12

Поддержка украинского шрифта.

ВАРИАНТ 13

При переходе на латиницу нажатие каждой клавиши подтверждается звуковым сигналом.

ВАРИАНТ 14

Настройка из командной строки цвета рамки, индицирующей кириллицу.

Примечание: Установка цвета рамки экрана VGA (прерывание 10h):
На входе: AH = 10h; AL = 01H – установка цвета рамки; BH = цвет.

ВАРИАНТ 15

Индикация режима работы драйвера строками Russian/English в верхнем правом углу экрана.

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА № 5
ТЕМА: ПРОГРАММИРОВАНИЕ ВИДЕОАДАПТЕРОВ

Цель: получение навыков в разработке процедур построения графических примитивов способом прямой записи в видеопамять

1. Методические указания к выполнению работы

Основой для видеоадаптеров служит микросхема CRTC (Cathode Ray Tube Controller) того или иного производителя. Эта микросхема выполняет массу технических задач, которые не интересуют программиста. Однако она также устанавливает режим работы экрана, управляет курсором и цветом. Микросхема легко программируется напрямую, хотя процедуры операционной системы позволяют управлять большинством ее действий.

1.1 Типы адаптеров

Монохромный адаптер имеет 4 килобайта памяти на плате, начиная с адреса B000:0000. Этой памяти хватает только для хранения одной 80-символьной страницы текста.

Цветной графический адаптер (CGA, Color Graphic Adapter) имеет 16 килобайт памяти на плате, начиная с адреса памяти B800:0000. Этого достаточно для отображения одного графического экрана без страниц, или для отображения от четырех до восьми экранов текста в зависимости от числа символов в строке – 40 или 80.

EGA (Enhanced Graphic Adapter) может быть снабжен 64, 128 или 256 килобайтами памяти. Кроме использования в качестве видеобуфера, эта память может хранить битовые описания до 1024 символов. Стартовый адрес буфера дисплея программируемый, поэтому буфер начинается с адреса A000:0000 для улучшенных графических режимов и с B000:0000 и B800:0000 для совместимости со стандартным монохромным и цветным графическим режимом.

VGA (Video Graphic Array) имеет 256 килобайт памяти на плате, которые разделены на четыре цветовых слоя по 64 килобайта. VGA почти полностью аналогичен EGA (включая плоскостную видеопамять в 16-ти цветных режимах и секвенсор для доступа процессора к ней).

Видеоадаптер SVGA имеет от 512 килобайт до 32 мегабайт видеопамяти. Большая часть видеопамяти может использоваться не только для хранения изображения, но и для хранения текстур в 3-х мерных играх.

1.2 Программирование контроллера

Все системы строятся вокруг микросхемы контроллера. Применение микросхемы во многом идентично в различных адаптерах. Контроллер устанавливает дисплей в один из текстовых или графических режимов, выполняет основную работу по интерпретации кодов ASCII и поиску данных для вывода соответствующих символов.

Микросхема контроллера имеет 18 управляющих регистров, пронумерованных от 0 до 17. Первые 10 фиксируют горизонтальные и вертикальные параметры дисплея. Эти регистры, как правило, неинтересны для программиста, поскольку они автоматически устанавливаются BIOS при изменении размера экрана. Эксперимент с данными регистрами может привести к физической порче дисплея.

Регистры имеют размер 8 бит, но некоторые из них связаны в пары, чтобы хранить 16-битовые значения. Пары 10 – 11 и 14 – 15 устанавливают форму и местоположение курсора. Пара 12 – 13 управляет страницами дисплея. Пара 16 – 17 сообщает позицию светового пера. У EGA и VGA есть еще 6 добавочных регистров, которые связаны с техническими деталями. Регистр 20 представляет определенный интерес: он определяет, какая линия сканирования в строке символа используется для подчеркивания.

Доступ ко всем 18 регистрам осуществляется через один и тот же порт, адрес которого для монохромного адаптера – 3B5h, для цветного – 3D5h. EGA и VGA используют один из этих двух адресов в зависимости от типа подключенного к нему монитора. Для записи в регистр необходимо вначале в регистр адреса (3B4h – для монохромного адаптера, 3D4h – для цветного) послать номер требуемого регистра. Тогда следующий байт, посланный в порт с адресом 3B5h или 3D5h, будет записан в этот регистр.

В нижеприведенном фрагменте кода программы курсор устанавливается в позицию экрана, указанную в регистре BX, записью содержимого BX в управляющие регистры 14 и 15 контроллера. BH – колонка экрана (в примере – 79), BL – строка экрана (в примере – 24).

```
; запись данных из BX в регистры 14 и 15 микросхемы 6845
mov bx,4F18H        ; 4FH = 79 и 18H = 24
mov ax,bx           ; копируем BX в AX
and ax,0ffh;        ; устанавливаем AX на строку (обнуляем AH)
mov cl,80           ; CL = количеству символов в строке
mul cl              ; AX = номер строки, умноженный на 80
mov dx,bx           ; устанавливаем DX на колонку
mov cl,8            ; счетчик сдвига
shr dx,cl           ; сдвигаем DH в DL
add ax,dx           ; добавляем к AX значение колонки
mov bx,ax           ; сохраняем позицию курсора в BX
; выбираем регистр младшего байта
mov dx,3D4h         ; порт регистра адреса
mov al,15           ; номер регистра для младшего байта регистра BX
out dx,al           ; посылаем номер регистра в порт 3D4h
; посылаем байт данных в управляющий регистр 15
mov dx,3D5h         ; выбираем порт для доступа к регистру 15
mov al,bl           ; берем младший байт регистра BX
out dx,al           ; посылаем его в регистр 15
```

```
; выбираем регистр старшего байта
mov dx,3D4h       ; порт регистра адреса
mov al,14         ; номер регистра для старшего байта регистра BX
out dx,al         ; посылаем номер регистра в порт 3D4h
; посылаем байт данных в управляющий регистр 14
mov dx,3D5h       ; выбираем порт для доступа к регистру 14
mov al,bh         ; берем старший байт
out dx,al         ; посылаем его в регистр 14
```

У монохромного и цветного адаптеров имеются еще три порта, которые важны для программистов. Они имеют адреса 3B8h, 3B9h и 3BAh для монохромного и 3D8h, 3D9h и 3DAh – для цветного адаптера. Первый устанавливает режим экрана, второй устанавливает цвета, а третий возвращает информацию о статусе дисплея.

1.3 Инициализация режима

В таблице 5.1 приведен перечень различных режимов адаптеров.

Таблица 5.1 – Перечень режимов адаптеров

Номер	Режим	Адаптеры
0	40x25, ч/б, текст.	CGA, EGA, VGA
1	40x25, текст.	CGA, EGA, VGA
2	80x25, ч/б, текст.	CGA, EGA, VGA
3	80x25, текст.	CGA, EGA, VGA
4	300x200, 4-цв., граф.	CGA, EGA
5	300x200, ч/б, граф.	CGA, EGA
6	640x200, ч/б, граф.	CGA, EGA
7	80x25, ч/б, текст.	MDA, EGA
8-A	Зарезервированы	
B-C	Зарезервированы	
D	300x200, 16-цв., граф.	EGA
E	640x200, 16-цв., граф.	EGA
F	640x350, 4-цв., граф.	EGA
10	640x350, 16-цв., граф.	EGA
11	640x480, 2-цв., граф.	VGA
12	640x480, 16-цв., граф.	VGA
13	320x200, 256-цв., граф.	SVGA
100h	640 x 400, 256-цв., граф.	SVGA
101h	640 x 480, 256-цв., граф.	SVGA
102h	800 x 600, 16-цв., граф.	SVGA
103h	800 x 600, 256-цв., граф.	SVGA
104h	1024 x 768, 16-цв., граф.	SVGA
105h	1024 x 768, 256-цв., граф.	SVGA
106h	1280 x 1024, 16-цв., граф.	SVGA
107h	1280 x 1024, 256-цв., граф.	SVGA

Для инициализации соответствующего режима необходимо вызвать функцию 0 прерывания 10h. Для этого необходимо поместить номер функции в AH, номер режима – в AL, и осуществить вызов прерывания.

Для установки видеорежима по стандарту VESA Super VGA также служит прерывание BIOS 10h, но в этом случае всё сложней. В регистр BX заносится номер видеорежима, причём, если старший бит в номере видеорежима выключен, то видеопамять очищается, если включён, то видеопамять не очищается. В регистр AH заносится номер функции для работы с видеоадаптером SVGA – 4Fh, а в регистр AL – номер подфункции для установки видеорежима – 02h. Вызывается программное прерывание int 10h и, если после этого в регистре AL будет 4Fh, то функция поддерживается BIOS. Если в AL будет 0, то операция установки видеорежима выполнена успешно, если 1, то произошла ошибка (либо у персональной ЭВМ нет видеоадаптера SVGA, либо он не поддерживает данный видеорежим).

Пример установки видеорежима SVGA 800 x 600 256 цветов с очисткой видеопамяти:

```
mov     ah, 4Fh
mov     al, 02h
mov     bx, 103h
int     10h
cmp     al, 4Fh
jne     not_supported
cmp     ah, 0
jnz     error
. . . . .
error:
. . . . .
not_supported:
. . . . .
```

1.4 Установка атрибутов/цветов символов

Когда дисплей установлен в текстовый режим в любой из видеосистем, каждой позиции символа на экране отводится два байта памяти. Первый байт содержит номер кода ASCII символа, а второй – атрибуты символа, коды которых приведены в таблице 5.2.

Таблица 5.2 – Цвета символа и его фона

Код	Цвет	Код	Цвет
0	Черный	8	Серый
1	Синий	9	Голубой
2	Зеленый	10	Светло-зеленый
3	Циан	11	Светлый циан
4	Красный	12	Светло-красный
5	Пурпурный	13	Светло-пурпурный
6	Коричневый	14	Желтый
7	Белый	15	Ярко-белый

Младшие три бита атрибутов (0 – 2) устанавливают цвет самого символа (бит 3 включает высокую интенсивность). Следующие три бита (4 – 6) устанавливают фон символа. Старший бит (7) включает и выключает мигание символа.

1.5 Режимы MGA

MGA (Monochrome Graphics Adapter, также известный, как Hercules) является вариацией монохромного адаптера MDA. Он использует тот же тип монитора, что и MDA, но в дополнение к текстовому режиму MGA имеет еще и один графический режим.

Текстовый режим MGA: см. «Режимы CGA».

Графический режим: MGA позволяет использовать один графический режим: двухцветный, с разрешением 720x348 пикселей.

1.6 Режимы CGA

Раскладка памяти в текстовых режимах представляет собой простую плоскость двухбайтовых пар (первый байт – символ, второй – атрибут), расположенных по порядку – первая пара представляет первый символ первой строки, вторая – второй символ второй строки и т.д.

Графические режимы имеют более сложное представление. Скан-линии с четными номерами расположены по адресу B800:0000, а скан-линии с нечетными адресами – по адресу B800:2000. Каждая скан-линия занимает 80 байт. Всего существует 200 скан-линий (т.е. строк). Каждый байт содержит 4 пикселя (рис. 5.1). Всего 16000 пикселей.

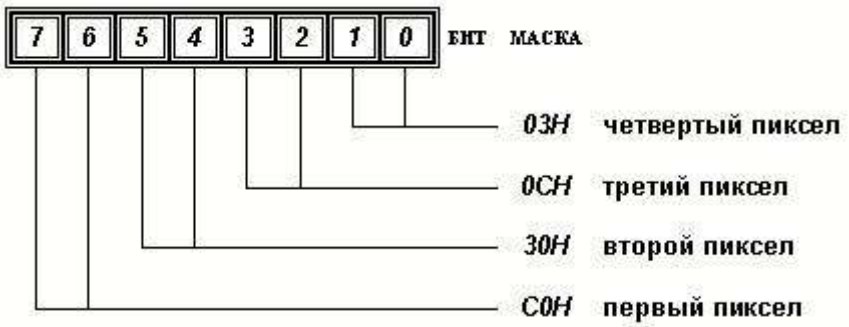


Рис.5.1 – расположение пикселей в байте видеопамяти в режиме CGA

Цвета: 00 = черный; 01 = зеленый/циан; 10 = красный/пурпурный; 11 = коричневый/белый.

Каждое 2-битовое поле выбирает один из четырех цветов, в зависимости от установленной палитры CGA.

1.7 Режимы EGA

У EGA-адаптера более сложная структура. С точки зрения процессора режимы экрана 0 – 7 действуют так же, как соответствующие режимы из CGA, но режимы от 0Dh до 10h кардинально от них отличаются.

В режимах 0Dh, 0Eh, 0Fh и 10h память разбита на четыре битовые плоскости. Каждая плоскость организована следующим образом: когда байт данных посылается в определенный адрес видеобуфера, каждый бит соответствует точке на экране, причем весь байт соответствует горизонтальному сегменту линии, а бит 7 – самой левой точке. Выводятся четыре такие битовые плоскости, относящиеся к одинаковым адресам в видеобуфере. Это приводит к тому, что каждая точка описывается 4 битами (что дает 16 цветов), причем каждый бит находится в отдельном байте отдельной битовой плоскости.

Изменение четырех различных байтов данных, расположенных по одному и тому же адресу, осуществляется следующим образом. Вы не посылаете последовательно четыре байта по этому адресу. Вместо этого один из трех режимов записи позволяет изменить все четыре байта на основании одного байта данных, полученного от процессора. Влияние данных, посланных процессором, зависит от установки нескольких регистров, включающих два регистра маски, которые определяют, на какие биты и в каких битовых плоскостях будут воздействовать изменения.

Для понимания этих регистров рассмотрим регистры задвижки («latch registers»). Они содержат данные для четырех битовых плоскостей в той позиции, к которой было последнее обращение. Заметим, что термин «битовая плоскость» используется как для целой области видеобуфера, так и для однобайтового буфера, временно хранящегося в регистре задвижки.

Когда процессор посылает данные по определенному адресу, эти данные могут изменить или полностью сменить данные регистра задвижки, а впоследствии *именно данные регистра задвижки записываются в видеобуфер*. Каким образом данные процессора влияют на регистр задвижки, зависит от режима записи, а также от значения некоторых других регистров. При чтении данных из видеобуфера регистры задвижки заполняются четырьмя байтами из четырех битовых плоскостей по данному адресу. Регистрами задвижки легко манипулировать, производя с их содержимым различные логические операции.

Регистр маски битов и *регистр маски карты* действуют на регистры задвижки, защищая определенные биты или битовые плоскости от изменения под действием данных, поступающих от процессоров.

Регистр маски битов – это регистр, предназначенный только для записи, адрес порта которого 3CFh. Сначала надо послать 8 в порт 3CEh, чтобы указать на этот регистр. Установка бита этого регистра в 1 маскирует этот бит во всех четырех битовых плоскостях, делая соответствующую точку недоступной для изменения. Необходимо перед записью по данному адресу, сначала считать из него. Это связано с техническими особенностями функционирования EGA-адаптера.

Регистр маски карты имеет адрес порта 3C5h. Этот регистр открыт только для записи. Перед посылкой данных в этот регистр надо послать по адресу 3C4h число 2 как указатель. Биты 0 – 3 этого регистра

соответствуют битовым плоскостям 0 – 3; старшие 4 бита не используются. Когда биты 0 – 3 равны 0, соответствующие битовые плоскости не изменяются при операциях записи. Это свойство используется по-разному в различных режимах записи.

1.8 Режимы записи

Три режима записи устанавливаются *регистром режима*, который является регистром «только для записи». Адрес порта регистра, – 3CFh. Регистр режима индексируется предварительной посылкой 5 в порт 3CEh. Режим записи устанавливается в битах 0-1 как число от 0 до 2. Бит 3 устанавливает один из двух режимов чтения из видеобуфера.

В простейшем случае *режим записи 0* копирует данные процессора в каждую из четырех битовых плоскостей. Пусть, например, по определенному адресу видеобуфера послано 11111111b и разрешены все биты и битовые плоскости. Тогда каждый бит во всех четырех плоскостях будет установлен в 1, так, что цепочка битов для каждой из соответствующих точек будет установлен в 1111b. Это означает, что 8 точек будут выведены в цвете 15 (ярко-белом).

Режим записи 1 предназначен для специальных приложений. В этом режиме текущее содержимое регистра задвижки записывается по указанному адресу. Стоит напомнить, что регистры задвижки заполняются операцией чтения. Этот режим очень полезен для быстрого переноса данных при операциях сдвига экрана. Регистр маски битов и регистр маски карты не влияют на эту операцию.

Режим записи 2 предоставляет альтернативный способ установки отдельных точек. Процессор посылает данные, у которых имеют значение только 4 младших бита, рассматриваемые как цвет. Можно сказать, что эта цепочка битов вставляется поперек битовых плоскостей. Цепочка дублируется на все восемь точек, относящихся к данному адресу, до тех пор, пока регистр маски битов не предохраняет определенные точки от изменения, Регистр маски карты активен, как и в режиме записи 0. Конечно, процессор должен послать полный байт, но только младшие четыре бита существенны.

В нижеприведенном фрагменте кода программы выводится красная точка в позицию экрана, определяемую значениями переменных X и Y в графическом режиме 12H адаптера VGA (640x480 точек, 16 цветов). В этом режиме точки одной строки экрана занимают в видеопамяти 640 точек / 8 бит = 80 байт. Смещение байта, содержащего точку с координатами X и Y, относительно начального адреса видеопамяти вычисляется по формуле:

*номер байта = Y * 80 + (целая часть от деления X на 8).*

Сдвиг бита цвета в байте равен остатку от деления X на 8.

Выбирается режим записи с номером 2 и используется регистр маски битов.

```
; устанавливаем графический режим 640x480, 16-цветов
mov   ah,0
mov   al,12h
int     10h

; устанавливаем режим записи 2
mov   dx,3ceh       ; указываем на регистр адреса
mov   al,5          ; индексируем регистр 5
out dx,al           ; посылаем индекс

mov dx,3cfh;        ; указываем на регистр режима
mov al,2            ; выбираем режим записи 2
out dx,al           ; устанавливаем режим

mov ax,0A000h       ; адрес начала видеобуфера
mov es,ax           ; помещаем в регистр es
; высчитываем по формуле
; номер байта = Y * 80 + (целая часть от деления X на 8)
; смещение байта, содержащего точку с координатам X и Y, относительно
; начального адреса видеопамяти и помещаем его в регистр bx
; . . . . . .
; высчитываем сдвиг бита цвета в байте как остаток от деления X на 8
; и помещаем его в регистр cl
; . . . . . .
; устанавливаем регистра маски битов
mov   dx,3ceh           ; указываем на адресный регистр
mov   al,8              ; регистр маски битов
out dx,al               ; адресуем регистр

mov   dx,3cfh           ; указываем на регистр данных
mov   al,10000000b      ; подготавливаем маску для смещения
shr     al,cl           ; сдвигаем маску вправо  на cl бит
out     dx,al           ; посылаем данные

; рисуем точку красного цвета
mov   al,es:[bx]        ; заполняем регистры задвижки
mov   al,4              ; устанавливаем цвет
mov   es:[bx],al        ; рисуем точку
```

1.9 Режимы чтения

Управляя регистром режима EGA, можно установить *два режима чтения. В режиме 0* возвращается байт, содержащийся во всех четырех битовых плоскостях, по указанному адресу. Режим 1 ищет указанный код цвета и возвращает байт, в котором бит установлен в 1, когда соответствующая точка имеет данный цвет. Доступ к этому регистру осуществляется через порт 3CFh, о работе с которым уже упоминалось.

Режим чтения 0 требует установки регистра выбора карты. Задача этого регистра – установить, какая из карт битов должна быть прочитана.

Режим чтения 1 более сложен. Сначала регистр сравнения цветов должен быть заполнен цепочкой битов для кода цвета, который вы ищете. Этот код помещается в младшие четыре бита регистра, старшие биты несущественны. (В порт 3CEh посылается 2, затем цепочка битов посылается в порт 3CFh.) После чтения ячейки памяти возвращается байт, у которого биты равны 1 для каждой точки, имеющей нужный цвет.

1.10 Обзор возможностей адаптеров VGA и SVGA

Адаптер VGA (Video Graphic Array, видеографический массив) был разработан фирмой IBM для применения в серии PS/2. Хотя его интерфейс и схож с EGA, он значительно отличается от своего предшественника. На выходе этот адаптер дает не цифровой, а аналоговый сигнал, что делает его несовместимым по типам дисплеев с EGA. Он способен одновременно отображать 256 цветов из 2^{18} цветов палитры. Его максимальное разрешение в графических режимах увеличилось до 640x480 пикселей.

Адаптер SVGA (Super VGA) введен ассоциацией VESA. Его разрешение от 800x600 до 1600x1200 пикселей, количество цветов до 2^{24}.

1.11 Используемые адреса в области данных BIOS

Таблица 5.3 – Используемые адреса в области данных BIOS

Адрес	Длина	Комментарий
0040:0049	1	Текущий видеорежим
0040:004A	2	Ширина экрана в колонках текста
0040:004C	2	Размер (в байтах) памяти, используемой текущим режимом
0040:004E	2	Смещение активной видеостраницы
0040:0050	16	Место курсора (8 пар байтов: нижний – колонка, верхний – столбец)
0040:0060	2	Размер/форма курсора
0040:0062	1	Номер активной видеостраницы

2. Контрольные вопросы по теме

1. Типы видеоадаптеров.
2. Что входит в понятие «программирование видеоадаптера»?
3. Регистры контроллера и доступ к ним.
4. Инициализация режимов видеоадаптера.
5. Структура байта атрибутов символа.
6. Характеристика режимов работы адаптеров MGA и CGA.
7. Характеристика основных режимов работы адаптера EGA.
8. Режимы записи 0,1,2 в режимах EGA (начиная с 0Dh).
9. Режимы чтения 0,1 для EGA.
10. Основные отличия адаптера VGA от EGA.
11. Используемые адреса в области данных BIOS.

3. Варианты индивидуальных заданий

Составить функцию построения графического примитива, в соответствии с номером варианта, методом прямой записи в видеопамять.

ВАРИАНТ 1

Построение пикселя с координатами (x, y).

ВАРИАНТ 2

Перемещение пикселя из положения (x1,y1) в положение (x2, y2) без отображения промежуточных положений.

ВАРИАНТ 3

Построение горизонтальной линии с началом в (x,y) и концом в (x+dx, y).

ВАРИАНТ 4

Построение произвольной ломаной линии.

ВАРИАНТ 5

Построение прямоугольника.

ВАРИАНТ 6

Построение любой русской буквы по точкам.

ВАРИАНТ 7

Построение штриховой линии.

ВАРИАНТ 8

Закраска прямоугольной области экрана.

ВАРИАНТ 9

Построение любой английской буквы по точкам.

ВАРИАНТ 10

Построение произвольной линии.

ВАРИАНТ 11

Построение окружности.

ВАРИАНТ 12

Построение вертикальной линии с началом в (x,y) и концом в (x+dx, y).

ВАРИАНТ 13

Перемещение точки по экрану с фиксацией всех промежуточных состояний.

ВАРИАНТ 14

Перемещение линии по экрану (горизонтальное, вертикальное).

ВАРИАНТ 15

Перемещение прямоугольника по экрану.

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА № 6
ТЕМА: ОРГАНИЗАЦИЯ ФАЙЛОВОЙ СИСТЕМЫ

Цель: изучение файловой системы. Получение навыков программирования процедур, использующих обращения к файловой системе.

1. Методические указания к выполнению работы

1.1. Организация дискового пространства

Жесткие диски, как правило, состоят из нескольких параллельных пластин, у каждой из которых есть по две головки, чтобы читать с обеих сторон. Поверхность диска распределена на ряд концентрических колец, называемых дорожками, а дорожки делятся радиально на сектора. Все дорожки, расположенные на одинаковом расстоянии от центра, называются цилиндром. Группы цилиндров (разделы) могут относиться к различным файловым системам.

Например, если абстрактный диск имеет одну пластину, т.е. две стороны (головки), на каждой из сторон – 80 дорожек, и каждая дорожка разбита на 18 секторов по 512 байт, то емкость такого диска составляет 2*80*18*0,5 Кб = 1,44 Мб.

Файл располагают на таком количестве секторов, которое необходимо, чтобы вместить его полностью. Только несколько секторов зарезервированы для служебных нужд. Остальные заполняются по принципу FIFO. Это означает, что по мере заполнения диска сектора заполняются по направлению к центру диска. При уничтожении файла сектора освобождаются и со временем свободные области становятся разбросанными по диску, разбивая новые файлы и замедляя к ним доступ.

Дисковые сектора определяются магнитной информацией, которую записывает утилита форматирования. Информация включает идентификационный номер каждого сектора. Дорожки не маркируются, вместо этого они определяются механически по смещению головки. Следует отметить, что функции операционной системы рассматривают все сектора диска как одну цепь, которая нумеруется подряд, начиная от 0.

Диски имеют главную загрузочную запись MBR (Master Boot Record), включающую таблицу разделов (Partition Table), которая позволяет распределить диск между несколькими файловыми системами. Таблица содержит информацию, где на диске начинается и заканчивается раздел.

1.2. Таблица размещения файлов (FAT, File Allocation Table)

Диск использует FAT для распределения дискового пространства между файлами и хранения информации о свободных секторах. Из соображений надежности на каждом диске хранится по две копии FAT. Они хранятся последовательно, в секторах с самыми младшими доступными логическими номерами, начиная со стороны 0, дорожки 0, сектора 2. Число секторов под FAT определяется размерами и типом диска.

Размер записи FAT для жесткого диска равен 16 или 32 битам (системы FAT16 и FAT32). Здесь рассмотрим более простые 12-битовые записи.

FAT хранит информацию о каждом кластере секторов на диске. Кластер – это группа стандартных секторов размером в 512 байт. Группировка секторов необходима, чтобы уменьшить размер FAT. Однако большие кластера, используемые на диске, приводят к напрасному расходованию дискового пространства при записи маленьких файлов.

Каждая позиция в таблице размещения файлов соответствует определенной позиции кластера на диске. Обычно файл занимает несколько кластеров и запись в каталоге файлов содержит номер стартового кластера, в котором находится начало файла. Просмотрев позицию FAT, соответствующую первому кластеру, можно найти номер кластера, в котором хранится следующая порция файла. Этому кластеру соответствует своя запись в FAT и т.д. Для последнего кластера, занятого файлом, FAT содержит значения от FF8h до FFFh. Свободным кластерам соответствует значение 000, а дефектным кластерам – FF7h. Резервным кластерам приписываются значения от FF0h до FF7h.

Номер кластера содержит три шестнадцатеричные цифры, для хранения которых требуется полтора байта. Таким образом, числа для двух последовательных кластеров хранятся в трех последовательных байтах.

Первые три байта FAT зарезервированы. Первый байт содержит код, определяющий тип диска, а следующие два равны FFh. Кластера нумеруются, начиная с 2. Кластерам 2 и 3 соответствует вторая тройка байтов таблицы.

1.3. Работа с каталогами диска

Каждый диск имеет один корневой каталог, с которого начинается поиск всех остальных каталогов. Корневой каталог может содержать элементы, указывающие на подкаталоги, которые в свою очередь могут содержать ссылки на другие подкаталоги, образуя древовидную структуру каталогов. Корневой каталог всегда расположен в определенных секторах диска. Подкаталоги хранятся, как обычные дисковые файлы и могут находиться в любом месте диска.

Как корневой каталог, так и подкаталоги используют 32 байта для хранения информации об одном файле, независимо от типа диска. Каждое 32-байтовое поле разбито следующим образом (таблица 6.1)

Таблица 6.1 – поля записи каталога

0 – 7	имя файла
8 – 10	расширение
11	атрибут
12 – 21	зарезервировано
22 – 23	время последнего доступа к файлу
24 – 25	дата последнего доступа к файлу
26 – 27	начальный кластер
28 – 31	размер файла

Точка между именем и расширением файла не хранится. Все поля выравнены по левой границе, а пустые байты заполняются пробелами (код 32). Атрибут определяет, является ли файл скрытым, защищенным от записи и т.п. Начальный кластер указывает на позицию в FAT. Поскольку файл обычно не занимает последний отведенный ему кластер полностью, то поле «размер файла» хранит точную длину файла.

1.4. Работа с файлами

В операционной системе DOS использует два метода доступа к файлам: метод FCB (File Control Block, Управляющий Блок Файла), и метод дескриптора файла (File Handle). Метод FCB существует с тех пор, когда операционная система еще не работала с древовидной структурой каталогов, поэтому с его помощью можно получить доступ только к файлам, находящимся в текущем каталоге. Метод дескриптора файла позволяет получить доступ к любому файлу.

Поскольку древовидная структура каталогов сейчас используется повсеместно, то метод FCB является анахронизмом. Поэтому необходимо применять в своих программах метод дескриптора файла. Метод дескриптора файла имеет дополнительное преимущество в том, что он требует меньше подготовительной работой.

Прежде чем читать или писать данные в файл, его нужно открыть, т.е. инициализировать специальную область данных, используемую операционной системой, которая содержит информацию о файле. Одной из таких областей является управляющий блок файла и, когда используется метод FCB, программа создает такой блок, а операционная система читает и манипулирует его содержимым. Первоначально FCB содержит только имя файла и имя накопителя, при открытии файла, в него добавляется информация о размере записи и текущей позиции.

При доступе к файлам с помощью дескриптора файла операционная система автоматически создает область данных для файла в произвольном месте. Затем операционная система создает уникальный 16-битовый код номера файла. Впоследствии этот номер используется функциями операционной системы для идентификации файла. Все, что нужно для нахождения файла – это стандартная строка пути, которая должна заканчиваться символом с кодом 0.

Функции доступа через управляющий блок файла определяют промежуточный буфер с помощью указателя, который все время хранится операционной системой. Этот буфер называется областью обмена с диском (DTA, Disk Transfer Area). По умолчанию эта область занимает 128 байт в префиксе программного сегмента (PSP) по смещению 80h. Если необходимо переопределить DTA, то следует использовать функцию 1Ah прерывания 21h. Перед ее вызовом необходимо установить DS:DX так, чтобы они указывали на первый байт DTA.

Структура FCB приведена в таблице 6.2.

Таблица 6.2 – Структура блока управления файлом

Смещение	Длина	Содержимое
0	1	Число, определяющее накопитель: 0 = по умолчанию, 1 = A, 2 = B и т.д.
1	8	Восьмибайтовое имя файла, дополненное справа пробелами
9	3	Трехбайтовое расширение, дополненное справа пробелами
0CH	2	Номер текущего блока. Блок файла содержит 128 записей, пронумерованных от 0 до 127. Например, запись №129 рассматривается как запись №0 блока №1. Для обращения к конкретной записи используется номер текущего блока и номер текущей записи (см. байт со смещением 20H). Первый блок файла имеет номер 0, второй - 1 и т.д. Операция открытия файла устанавливает в данном поле 0
0EH	2	Размер логической записи. По умолчанию 128 байт
10H	4	Размер файла с точностью до байта. Устанавливается после открытия файла
14H	2	Дата последней модификации файла
16H	10	Зарезервировано
20H	1	Относительный номер записи в блоке. Данное поле содержит текущий номер записи (0 –127) в текущем блоке. Операционная система использует текущие значения блока и записи для локализации записи в дисковом файле. Обычно номер начальной записи в данном поле – 0, но его можно заменить на любое значение от 0 до 127
21H	4	Относительный номер записи в файле. Используется для произвольного доступа к записи при операциях чтения или записи. Данное поле должно содержать относительный номер записи. Например, для чтения записи номер 25 (19H), без предварительного чтения первых 24 записей, необходимо установить в данном поле значение 19000000. В случае произвольного доступа операционная система автоматически преобразует относительный номер записи в текущие номера блока и записи
25H	1	Длина открытого FCB (используется только при размере записи меньше 64)

Расширенный FCB на 7 байт длиннее, причем эти 7 байт предшествуют обычному блоку. Расширенный FCB необходим для создания или доступа к файлу, имеющему специальные атрибуты, например, к скрытому (таблица 6.3).

Таблица 6.3 – Поля расширенного блока управления файлом

Смещение	Длина	Содержимое
-7	1	0FFH – флаг расширения FCB
-6	5	Зарезервировано
-1	1	Атрибут файла

Пример описания блока FCB:

```
FILE     LABEL   BYTE       ; список параметров
FCBDRV   DB      03         ; диск C
FCBNAME  DB      'PROBA   ' ; имя файла, дополненное
                            ; тремя пробелами
FCBEXT   DB   'DAT'         ; расширение
FCBBLK   DW   0000          ; номер текущего блока
FCBRCSZ  DW   ?             ; размер логической записи.
         DD   ?             ; размер файла
         DW   ?             ; дата последней модификации
         DT   ?             ; зарезервировано
FCBSQRC  DB   00            ; номер текущей записи.
FCBOTRC  DD   ?             ; относительный номер записи в файле.
```

2. Контрольные вопросы по теме

1. Физическая организация дискового пространства.
2. Организация и назначения FAT. Понятие «кластер».
3. Структура элемента каталога.
4. Структура дискового файла в терминах «блок» и «запись».
5. Метод FCB и метод дескриптора доступа к файлу.
6. Назначение DTA. Способы задания DTA.
7. Структура FCB.
8. В каких случаях применяется расширенный FCB?

3. Варианты индивидуальных заданий

Информацию для работы программы передавать в командной строке.

ВАРИАНТ 1

Написать программу копирования файла. Используемый метод: FCB, последовательный способ доступа к записям файла (используются байты 0CH, ODH и 20H). Переопределить DTA.

ВАРИАНТ 2

Написать программу копирования файла. Используемый метод: FCB, прямой способ доступа к записям файла (используются байты 21H, 22H, 23H и 24H). Переопределить DTA.

ВАРИАНТ 3

Написать программу копирования файла. Используемый метод: FCB, последовательный способ доступа к записям файла (используются байты 0CH, ODH и 20H). Для рабочих областей использовать область PSP.

ВАРИАНТ 4

Написать программу копирования файла. Используемый метод: FCB, прямой способ доступа к записям файла (используются байты 21H, 22H, 23H и 24H). Для рабочих областей использовать область PSP.

ВАРИАНТ 5

Написать программу копирования файла (метод дескриптора файла).

ВАРИАНТ 6

Написать программу, позволяющую выполнять следующие операции с каталогами: создание каталога; уничтожение каталога; изменение текущего каталога; получение имени текущего каталога;

ВАРИАНТ 7

Написать программу копирования файла, используя метод дескриптора файла. В качестве входного параметра указывать лишь имя копируемого файла. Программа должна сообщать имя получаемого файла, генерируемое операционной системой.

ВАРИАНТ 8

Написать программу перемещения файла, используя метод дескриптора файла. Перемещение должно осуществляться путём копирования с последующим удалением исходного файла.

ВАРИАНТ 9

Написать программу, позволяющую выполнять следующие операции с накопителями: получение имени текущего накопителя; установка текущего накопителя; получение имени текущего каталога накопителя.

ВАРИАНТ 10

Обработать символьный файл: если его длина меньше N байт, то дополнить его пробелами до размера N; иначе – урезать до размера N.

ВАРИАНТ 11

Реализовать программно команду операционной системы DIR.

ВАРИАНТ 12

Написать программу перемещения файлов из одного заполненного каталога в другой пустой каталог.

ВАРИАНТ 13

Реализовать программно команду DEL операционной системы. В имени файла может использоваться джокер «?».

ВАРИАНТ 14

Обработать символьный файл, заменив в нем все пробелы на символ подчеркивания.

ВАРИАНТ 15

Реализовать программно команду DOS ATTRIB.

ВАРИАНТ 16

Написать программу удаления заполненного каталога.

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА №7
ТЕМА: РАЗРАБОТКА КОМПИЛЯТОРА

Цель: знакомство со структурой и функциями компиляторов

1. Методические указания к выполнению работы
1.1 Общая схема компиляции

Приблизительная схема компиляции показана на рис. 7.1.

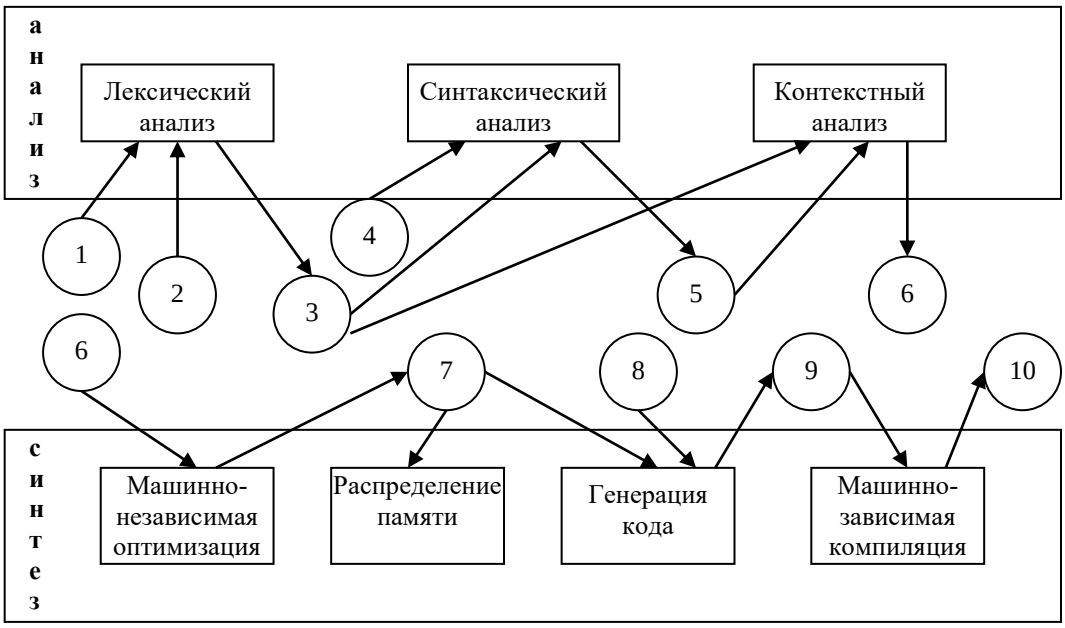


Рис. 7.1 – Общая схема компиляции

1) исходный текст программы;

2) таблица терминальных символов языка (*if, then, while* и т.д.);

3) лексическая свертка программы – получение промежуточной формы программы, удобной для работы синтаксического анализатора;

4) правила грамматики – правила редукции, аксиомы грамматики;

5) дерево вывода – дерево разбора программы;

6) абстрактная программа – атрибутное дерево программы. Отличие атрибутного дерева от простого дерева программы: узлы снабжаются атрибутами (могут быть ссылки на таблицу имен, ссылки к другим узлам атрибутного дерева, конкретные вычисленные значения);

7) некоторая промежуточная форма программы после машинно-независимой оптимизации. Может быть, несколько этапов такой оптимизации и несколько форм представления промежуточного машинного кода;

8) таблица констант – содержит кодовые продукции или заготовки, шаблоны, по которым строится следующая версия промежуточной формы 9;

9) команды Ассемблера или команды, напоминающие ассемблерные. Возможно, для кода 9 еще не было аппаратного распределения регистров.

Для облегчения построения компиляторов язык высокого уровня описывается в терминах некоторой *грамматики*. Эта грамматика определяет форму (*синтаксис*) допустимых предложений языка.

Например, оператор присваивания может быть определён в грамматике как имя переменной, за которой следует оператор присваивания (:=), за которым следует выражение. Проблема компиляции – проблема поиска соответствия написанных программистом предложений структурам, определённым грамматикой, и генерации соответствующего кода для каждого предложения.

Изображённая на рис. 7.2 программа на Паскале представлена в качестве иллюстрирующего примера при рассмотрении операций, необходимых для компиляции программ языка высокого уровня. Но ниже рассмотренные подходы и концепции применимы также к компиляции программ на других языках.

```
1     PROGRAM STATS;
2     VAR
3        SUM, SUMSQ, I, VALUE, MEAN, VARIANCE : INTEGER
4     BEGIN
5          SUM := 0;
6          SUMSQ := 0;
7          FOR I := 1 TO 100 DO
8               BEGIN
9               READ(VALUE);
10              SUM := SUM + VALUE;
11              SUMSQ := SUMSQ + VALUE * VALUE
12              END;
13         MEAN := SUM DIV 100;
14         VARIANCE := SUMSQ DIV 100 - MEAN * MEAN;
15         WRITE(MEAN, VARIANCE)
16    END.
```

Рис. 7.2 – Пример программы на Паскале

Основных шагов при компиляции программ – три:

1) лексический анализ;

2) синтаксический анализ;

3) генерация кодов.

Предложения исходной программы удобнее представлять в виде последовательности *лексем* (*tokens*), чем просто как строку символов. Лексемы можно понимать как фундаментальные кирпичики, из которых строится язык. Например, лексемой может быть ключевое слово, имя целой переменной, арифметический оператор и т.д. *Лексическим анализом называются просмотр исходного текста, распознавание и классификация различных лексем*. Часть компилятора, которая выполняет эту функцию, называют *лексическим анализатором (scanner).*

Как только лексемы выделены, каждое предложение программы может быть распознано как некоторая конструкция языка, как, например,

декларативные операторы или оператор присваивания, описанные с помощью грамматики. Процесс, называемый *синтаксическим анализом* или *синтаксическим разбором*, осуществляется той частью компилятора, которая называется *синтаксическим анализатором* (*parser*).

Последним шагом схемы процесса трансляции является *генерация объектного кода*. Большинство компиляторов генерирует непосредственно программу в машинных кодах, а не программу на Ассемблере, предназначенную для последующей трансляции в машинные коды.

Хотя основных шагов при компиляции программ – три, компилятор не обязательно делает три просмотра транслируемой программы. Для одних языков возможна компиляция программы за один просмотр, для других характерны многопросмотровые компиляторы (например, компиляторы, выполняющие оптимизацию объектного кода или какой – либо другой анализ программы).

1.2 Грамматики

Грамматика языка программирования является формальным описанием его *синтаксиса* или формы, в которой записаны отдельные предложения программы или вся программа. Грамматика не описывает *семантику* или значения различных предложений. Информация о семантике содержится в программах генерации объектного кода. В качестве иллюстрации разницы между синтаксисом и семантикой рассмотрим два предложения:

I := J + K и I := X + Y,

где X и Y являются вещественными переменными, а I, J, K – целыми переменными. Эти два предложения имеют одинаковый синтаксис. Оба являются операторами присваивания, в которых присваиваемое значение определяется выражением, состоящим из двух имён переменных, разделённых оператором сложения. Однако семантика этих двух предложений совершенно различна. Первое предложение говорит о том, что переменные в выражении должны быть сложены с использованием целых арифметических операций, а результат сложения должен быть присвоен переменной I. Второе предложение задаёт сложение с плавающей точкой, результат которого должен быть преобразован в целое число перед присваиванием. Эти два предложения будут скомпилированы в совершенно различные последовательности машинных команд, хотя их грамматическое описание одинаково. Различия между ними проявятся на этапе генерации объектного кода.

Существует различные формы записи грамматик, например, форма Бекуса-Наура (БНФ). Хотя БНФ не является вполне адекватной для описания некоторых реально существующих языков программирования, однако, эта форма достаточно проста, широко используется и предоставляет достаточные средства для большинства приложений. На рис. 7.3 изображена одна из возможных грамматик БНФ для узкого подмножества языка Паскаль.

```
1     <prog> ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END.
2     <prog-name> ::= id
3     <dec-list>  ::= <dec> | <dec-list> ; <dec>
4     <dec>       ::= <id-list> : <type>
5     <type>      ::= INTEGER
6     <id-list>   ::= id | <id-list>, id
7     <stmt-list> ::= <stmt> | <stmt-list> ; <stmt>
8     <stmt>      ::= <assign> | <read> | <write> | <for>
9     <assign>    ::= id := <exp>
10    <exp>       ::= <term> | <exp> + <term> | <exp> - <term>
11    <term>      ::= <factor> | <term> * <factor> | <term> DIV <factor>
12    <factor>    ::= id | int | ( <exp> )
13    <read>      ::= READ ( <id-list> )
14    <write>     ::= WRITE ( <id-list> )
15    <for>       ::= FOR <index-exp> DO <body>
16    <index-exp> ::= id := <exp> TO <exp>
17    <body>      ::= <stmt> | BEGIN <stmt-list> END
```

Рис. 7.3. – Упрощённая грамматика Паскаля

Грамматика БНФ состоит из множества *правил вывода*, каждое из которых определяет синтаксис некоторой конструкции языка программирования. Например, правило 13 на рис. 7.3

<read> :: = READ (<id-list>)

определяет синтаксис предложения READ языка Паскаль, обозначенный в грамматике как <read>. Символ ::= можно читать как «является по определению». С левой стороны от этого символа находится определяемая конструкция языка (<read>), а с правой – описание синтаксиса этой конструкции. Строки символов, заключённые в угловые скобки <и>, называются *нетерминальными символами* (т.е. являются именами конструкций, определёнными внутри грамматики). То, что не заключено в угловые скобки, называется *терминальными символами* грамматики (лексемами). В этом правиле вывода нетерминальными символами являются <read> и <id-list>. Терминальными символами являются лексемы READ, (,). Таким образом, это правило определяет, что конструкция <read> состоит из лексемы READ, за которой следует лексема (, за ней следует конструкция языка, называемая <id-list>, за которой следует лексема). Пробелы при описании грамматических правил не существенны и вставляются только для наглядности.

Для распознавания нетерминального символа <read> необходимо чтобы было определение для нетерминального символа <id-list>. Это определение даётся правилом 6 на рис. 7.3:

<id-list> :: = id | <id-list>, id

Это правило предлагает две возможности, разделённые символом |, для синтаксиса нетерминального символа <id-list>. Первая возможность состоит в том, что <id-list> состоит просто из лексемы **id** (запись **id** означает идентификатор, распознаваемый сканером). Другой вариант состоит в том, что <id-list>, за которым следует лексема «,», за которой

следует лексема **id**. Правило является рекурсивным. Это означает, что конструкция <id-list> определяется фактически в терминах себя самой. В соответствии с этим правилом нетерминальным символом <id-list> является любая последовательность из одной или более лексем **id**, разделённых запятыми. Таким образом,

ALPHA является нетерминальным символом <id-list>, состоящим из единственного идентификатора ALPHA;

ALPHA, BETA также являются конструкцией <id-list>, состоящей из конструкции <id-list> ALPHA, за которой следует «,», за которой идёт идентификатор BETA и т.д.

Результат анализа исходного предложения в терминах грамматических конструкций удобно представлять в виде дерева. Такие деревья обычно называют *деревьями грамматического разбора или синтаксическими деревьями*. На рис. 7.4 (а) изображено дерево грамматического разбора для предложения READ(VALUE) с использованием правил вывода 13 и 6.

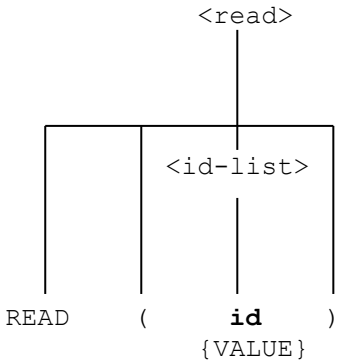


Рис.7.4 (а) – дерево грамматического разбора для предложения READ (VALUE)

Правило вывода 9 на рис. 7.3 даёт определение синтаксиса предложения присваивания:

<assign> :: = id := <exp>

Это означает, что конструкция <assign> состоит из лексемы id, за которой следует лексема :=, за которой идёт конструкция <exp>. Правило 10 даёт определение конструкции <exp>:

<exp> ::= <term> | <exp> + <term> | <exp> – <term>

Это правило определяет конструкцию <exp> как состоящую из любой последовательности конструкций <term>, соединённых операторами плюс или минус. Аналогично, правило 11 определяет конструкцию <term> как последовательность конструкций <factor>, разделённых знаками * или DIV. В соответствии с правилом 12 конструкция <factor> может состоять из идентификатора id или целого int, которое также распознаётся сканером, или из конструкции <exp>, заключённой в круглые скобки.

На рис. 7.4 (б) изображено дерево грамматического разбора для предложения 14 на рис. 7.2.

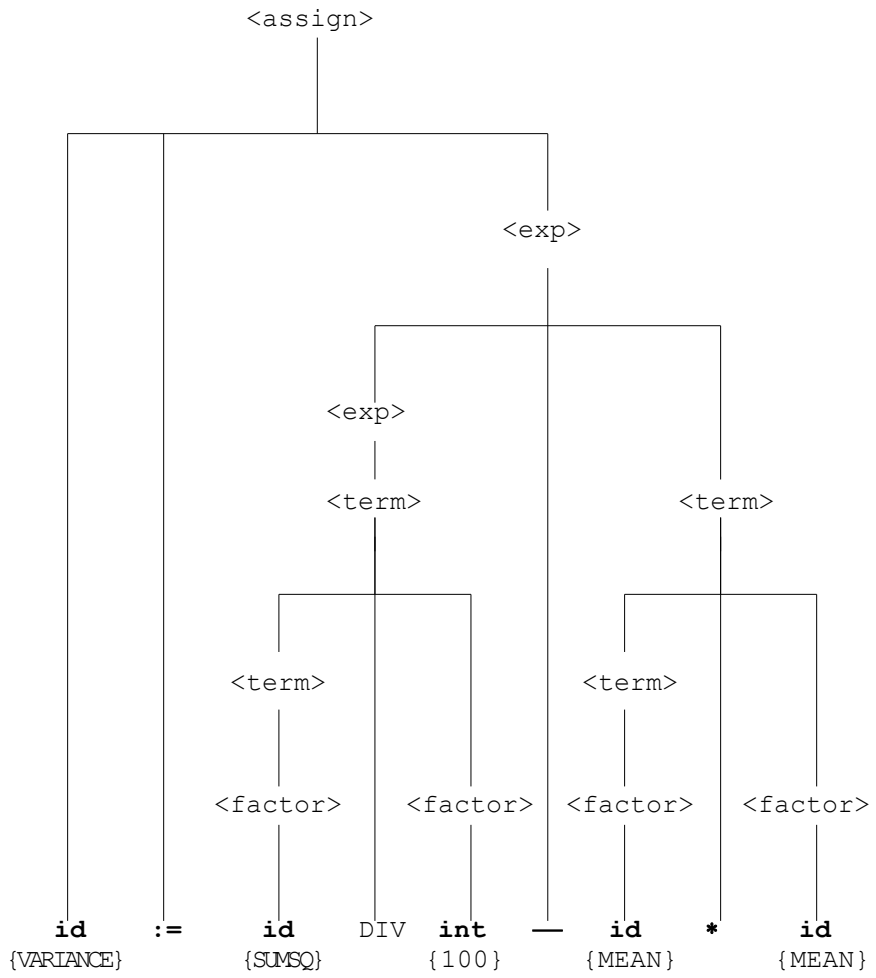


Рис. 7.4(б) – Дерево грамматического разбора для двух предложений программы на рис. 7.2

В соответствии с деревом грамматического разбора на рис 7.4 (б) умножение и деление производятся перед сложением и вычитанием. Прежде всего, должны вычисляться термы SUMSQ DIV 100 и MEAN * MEAN, поскольку эти промежуточные результаты являются операндами (левым и правым поддеревом) операции вычитания. То есть операции умножения и деления имеют более высокий ранг (*precedence*), чем операции сложения и вычитания. Такое ранжирование может быть использовано при осуществлении процесса грамматического разбора.

Дерево грамматического разбора на рис. 7.4(б) представляет собой единственно возможный результат анализа этих двух предложений в терминах грамматики, изображённой на рис. 7.3. Для некоторых грамматик подобной единственности может не существовать. Если для одного и того же предложения можно построить несколько различных деревьев грамматического разбора, то соответствующая грамматика называется *неоднозначной* (*ambiguous*). При разработке компиляторов обычно предпочитают пользоваться однозначными грамматиками.

1.3 Лексический анализ

Лексический анализ включает в себя сканирование компилируемой программы и распознавание лексем, составляющих предложения исходного текста. Сканеры обычно строятся таким образом, чтобы они могли распознавать *ключевые слова, операторы и идентификаторы так же, как целые числа, числа с плавающей точкой, строки символов и другие аналогичные конструкции*, встречающиеся в исходной программе. Точный перечень лексем, которые необходимо распознавать зависит от языка программирования, на который рассчитан компилятор и от грамматики, используемой для описания этого языка. Такие объекты, как идентификаторы и целые числа, обычно распознаются сканером как отдельные лексемы. Однако возможен и другой подход, в рамках которого эти лексемы описываются правилами грамматики. Например, идентификатор может быть определён следующим набором правил:

<ident> ::= <letter> | <ident> <letter> | <ident> <digit>
<letter> ::= A | B | C | D|... | Z
<digit> ::= 0 | 1 | 2 | 3 |... |9

Рис. 7.5 – Варианты определения идентификатора.

В этом случае сканер будет распознавать в качестве отдельных лексем единичные символы A, B, 0, 1 и т. д. Далее в процессе грамматического разбора последовательность этих символов будет интерпретирована как конструкция языка <ident>.

Однако в рамках такого подхода распознавание простых идентификаторов должны осуществлять общие алгоритмы грамматического разбора. Специализированные программы сканера могут выполнить эту функцию более эффективно. Поскольку основная часть программы состоит из подобных многосимвольных идентификаторов, сокращение времени компиляции может быть весьма существенным. Кроме того, такие ограничения, как максимальная длина идентификатора, гораздо легче учесть в сканере, чем в общих алгоритмах грамматического разбора.

Аналогично, сканер обычно распознаёт непосредственно как односимвольные, так и многосимвольные лексемы. Например, строку символов READ предпочтительнее рассматривать как отдельную лексему, нежели чем последовательность, состоящую из четырёх лексем R, E, A, D. Строка := будет распознана как оператор присваивания, а не как символ :, за которым следует =.

Возможен и такой подход, когда многосимвольные лексемы рассматриваются как состоящие из отдельных лексем – символов, но это существенно усложняет процесс грамматического разбора.

Результатом работы сканера является последовательность лексем. Для повышения эффективности последующих действий каждая лексема

обычно представляется некоторым кодом фиксированной длины (например, целым числом), а не в виде строки символов переменной длины. При подобной записи для грамматики, приведенной на рис. 7.3 лексеме *PROGRAM* соответствует целое число 1, идентификатору *id* соответствует число 22 и т.д.

Лексема	**Код**	**Лексема**	**Код**	**Лексема**	**Код**	**Лексема**	**Код**
PROGRAM	1	FOR	7	:	13	DIV	19
VAR	2	READ	8	,	14	(	20
BEGIN	3	WRITE	9	:=	15	)	21
END	4	TO	10	+	16	id	22
END.	5	DO	11	–	17	int	23
INTEGER	6	;	12	*	18		

Рис.7.6 – Коды лексем для грамматики на рис. 7.3

Если распознанная лексема является ключевым словом или оператором, такая схема кодирования даёт всю необходимую информацию.

В случае же идентификатора дополнительно необходимо конкретное имя распознанного идентификатора. То же относится к целым числам, числам с плавающей точкой, строкам символов и т.д. Этого можно добиться за счёт хранения не только кода лексемы соответствующего типа, но и дополнительного *спецификатора лексемы*. Спецификатор должен содержать имя идентификатора, значение целого числа и т.д. – всю информацию, распознанную сканером. Некоторые сканеры устроены таким образом, что записывают идентификатор, когда он встретился впервые, в таблицу символов, аналогичную таблице символов, используемой в Ассемблере. В этом случае спецификатором лексем – идентификаторов может служить указатель на соответствующий элемент таблицы символов, что позволяет избежать дополнительного поиска по таблицам на последующих этапах компиляции.

На рис. 7.7 изображен результат обработки сканером программы, приведенной на рис. 7.2, с использованием кодировки лексем, представленной на рис. 7.6. Для лексем типа 22 (идентификаторы) соответствующими спецификаторами являются указатели на таблицу символов (обозначаемые ^SUM, ^SUMSQ и т.д.). Для лексем, имеющих тип 23 (целые числа), спецификаторами являются значения соответствующих чисел (обозначаемые #0, #100 и т.д.). В данном случае результат работы сканера – список кодов лексем.

Строка	Тип лексемы	Спецификатор лексемы	Строка	Тип лексемы	Спецификатор лексемы
1	2	3	4	5	6
1	1 22	 ^STATS	10	22 15 22 16 22 12	^SUM ^SUM ^VALUE
2	2				
3	22 14 22 14 22 14 22 14 22 14 22 13 6	^SUM ^SUMSQ ^I ^VALUE ^MEAN ^VARIANCE 	11	22 15 22 16 22 18 22	^SUMSQ ^SUMSQ ^VALUE ^VALUE
4	3		12	4	
5	22 15 23 12	^SUM #0 	13	22 15 22 19 23 12	^MEAN ^SUM #100
6	22 15 23 12	^SUMSQ #0 	14	22 15 22 19 23 17 22 18 22 12	^VARIANCE ^SUMSQ #100 ^MEAN ^MEAN
7	7 22 15 23 10 23 11	 ^I #1 #100 	15	9 20 22 14 22 21	 ^MEAN ^VARIANCE
8	3				
9	8 20 22 21 12	 ^VALUE 	16	5	

Рис. 7.7. Результат лексического разбора программы на рис 7.2

Однако сканер может и не обрабатывать целиком всю программу до начала каких-либо других действий. Иногда сканер является процедурой, вызываемой в процессе грамматического разбора для получения очередной

лексемы. Тогда результатом каждого вызова сканера будет код очередной лексемы исходной программы (и, если необходимо, её спецификатор). При этом задача сохранения всей информации о лексемах, которая может понадобиться впоследствии, ложится на процесс грамматического разбора.

В дополнение к своей основной функции – распознаванию лексем – сканер обычно также выполняет чтение строк исходной программы и, возможно, печать листинга исходной программы. Комментарии игнорируются сканером, за исключением того случая, когда они должны быть напечатаны и, таким образом, эффективно удаляются из исходной программы до начала процесса грамматического разбора.

Правила распознавания лексем могут различаться в разных частях программы. Так, например, фрагмент READ не должен распознаваться в качестве ключевого слова, если он встречается внутри строки символов, заключённой в кавычки. Пробелы же оказываются существенными именно внутри таких закавыченных строк, даже если они не существенны в остальных частях программы.

1.4 Синтаксический анализ

Во время синтаксического анализа предложения исходной программы распознаются как языковые конструкции, описываемые используемой грамматикой. Этот процесс можно рассматривать как построение дерева грамматического разбора для транслируемых предложений. Методы грамматического разбора можно разбить на два класса – *восходящие* и *нисходящие* – в соответствии с порядком построения дерева. Нисходящие методы (методы сверху вниз) начинают с правила грамматики, определяющую конечную цель анализа с корня дерева грамматического разбора, и пытаются так его наращивать, чтобы последующие узлы дерева соответствовали синтаксису анализируемого предложения. Восходящие методы (методы снизу вверх) начинают с конечных узлов дерева грамматического разбора и пытаются объединить их построением узлов всё более и более высокого уровня до тех пор, пока не будет достигнут корень дерева.

Восходящий метод грамматического разбора называется методом *операторного предшествования*. Он основан на анализе пар последовательно расположенных операторов исходной программы и решении вопроса о том, какой из них должен выполняться первым.

Пример. Пусть необходимо вычислить арифметическое выражение:

A + B * C – D

В соответствии с обычными правилами арифметики умножение и деление осуществляются до сложения и вычитания, то есть умножение и деление имеют более высокий уровень *предшествования*, чем сложение и вычитание. При анализе первых двух операторов (+ и *) выяснится, что оператор + имеет более низкий уровень предшествования, чем оператор *. Часто это записывают следующим образом: **+ <• ***

Аналогично для следующей пары операторов (* и –) оператор * имеет более высокий уровень предшествования, чем оператор –. Можно записать это в виде: *** •> –.**

Метод операторного предшествования использует подобные отношения между операторами для управления процессом грамматического разбора. В частности, для рассмотренного арифметического выражения получим следующие отношения предшествования:

```
A + B * C – D
    <•   • >
```

Следовательно, подвыражение B * C должно быть вычислено до обработки любых других операторов рассматриваемого выражения. В терминах дерева грамматического разбора это означает, что операция * расположена на более низком уровне узлов дерева, чем операции + или –. Таким образом, рассматриваемый метод грамматического разбора должен распознать конструкцию B * C, интерпретируя её в терминах заданной грамматики, до анализа соседних термов предложения.

В рамках метода грамматического разбора, построенного на отношениях операторного предшествования, анализируемое предложение сканируется слева направо до тех пор, пока не будет найдено подвыражение, операторы которого имеют более высокий уровень предшествования, чем соседние операторы. Далее это подвыражение распознаётся в терминах правил вывода используемой грамматики. Этот процесс продолжается до тех пор, пока не будет достигнут корень дерева, что и будет означать окончание процесса грамматического разбора.

Первым шагом при разработке процессора грамматического разбора, основанного на методе операторного предшествования, должно быть установление отношений предшествования между операторами грамматики. При этом под *оператором* понимается любой терминальный символ (т.е. любая лексема). Таким образом, необходимо, в частности, установить отношение предшествования между лексемами BEGIN, READ, **id** и (.

Матрица на рис. 7.8 задаёт отношения предшествования для грамматики, приведенной на рис. 7.3.

Каждая клетка этой матрицы определяет отношение предшествования (если оно существует) между лексемами, соответствующими строке и столбцу, на пересечении которых находится эта клетка. Например:

```
PROGRAM ≐ VAR
и
BEGIN <• FOR
```

Отношение $\doteq$ означает, что обе лексемы имеют одинаковый уровень предшествования и должны рассматриваться грамматическим процессором в качестве составляющих одной конструкции языка.

	VAR	BEGIN	END	END.	INTEGER	FOR	READ	WRITE	TO	DO	;	:	,	:=	+	-	*	DIV	(	)	id	int
PROGRAM	≐																				<·	
VAR		≐									<·	<·	<·								<·	
BEGIN			≐	≐		<·	<·	<·			<·										<·	
END			·>	·>							·>											
INTEGER		·>									·>											
FOR																					<·	
READ																			≐			
WRITE																			≐			
TO										·>					<·	<·	<·	<·	<·		<·	<·
DO		<·	·>	·>		<·	<·	<·			·>										<·	
;		·>	·>	·>		<·	<·	<·			·>	<·	<·								<·	
:					<·																	
,																					≐	
:=			·>	·>					≐		·>				<·	<·	<·	<·	<·		<·	<·
+			·>	·>					·>	·>	·>				·>	·>	<·	<·	<·	·>	<·	<·
-			·>	·>					·>	·>	·>				·>	·>	<·	<·	<·	·>	<·	<·
*			·>	·>					·>	·>	·>				·>	·>	·>	·>	<·	·>	<·	<·
DIV			·>	·>					·>	·>	·>				·>	·>	·>	·>	<·	·>	<·	<·
(													<·		<·	<·	<·	<·	≐	<·	<·	<·
)			·>	·>					·>	·>	·>				·>	·>	·>	·>		·>		
id	·>		·>	·>					·>	·>	·>	·>	·>	≐	·>	·>	·>	·>		·>		
int			·>	·>					·>	·>	·>				·>	·>	·>	·>		·>		

Рис. 7.8 – Матрица предшествования для грамматики на рис. 7.3

Для многих пар лексем отношение предшествования не существует. Это означает, что соответствующие пары лексем не могут находиться рядом ни в каком грамматически правильном предложении. Подобная комбинация лексем должна рассматриваться как синтаксическая ошибка.

Существуют алгоритмы автоматического построения матриц предшествования, подобных изображенной на рис. 7.8. Для применимости метода операторного предшествования необходимо, чтобы отношения предшествования были заданы однозначно. Например, не должно быть одновременно отношений ; <• BEGIN и ; •> BEGIN.

На рис. 7.9(а) изображен результат применения метода операторного предшествования к двум предложениям программы рис. 7.2. На рис. 7.9 (а) изображен анализ предложения READ строки 9 этой программы. Это предложение анализируется по лексемам слева направо. Для каждой пары соседних операторов определено отношение предшествования. В строке (i) на рис. 7.9 (а) процессор грамматического разбора выделил фрагмент, ограниченный отношениями <• и •>, для распознавания в терминах

грамматики. В данном случае этот фрагмент содержит единственную лексему **id**. Эта лексема может быть распознана как нетерминальный символ <factor> в соответствии с грамматическим правилом 12. Однако эта лексема может быть также распознана как нетерминальные символы <prog-name> (правило 2) и <id-list> (правило 6). Для рассматриваемого метода не важно, какой конкретно нетерминальный символ распознан. Лексема **id** интерпретируется просто как некоторый нетерминальный символ $<N_1>$. В строке (ii) на рис. 7.9 (а) изображен результат анализа предложения, в котором лексема **id** заменена на $<N_1>$. Из него следует, что далее надо распознавать правый фрагмент предложения.

Строка (ii) на рис. 7.9 (а) изображает отношение предшествования для новой версии анализируемого предложения. Процессор грамматического разбора, основанный на отношении предшествования, обычно использует стек для хранения полученных от сканера лексем для их последующего распознавания. Отношения предшествования определены лишь для терминальных символов. Таким образом, нетерминальный символ $<N_1>$ не будет вовлечен в этот процесс, и далее отношения предшествования будут установлены между терминальными символами (и). В данном случае для последующего распознавания будет выделен фрагмент READ ($<N_1>$) который соответствует, за исключением имени нетерминального символа, грамматическому правилу 13. Это правило является единственным применимым для этого фрагмента. Обозначим этот фрагмент как нетерминальный символ $<N_2>$.

Разбор предложения READ закончен. Дерево грамматического разбора на рис. 7.4 совпадает с построенным, за исключением используемых имён нетерминальных символов. Это означает, что синтаксис анализируемого предложения определён правильно, а это и является целью процесса грамматического разбора. Имена нетерминальных символов были выбраны произвольно и не имеют отношения к смыслу предложений исходной программы.

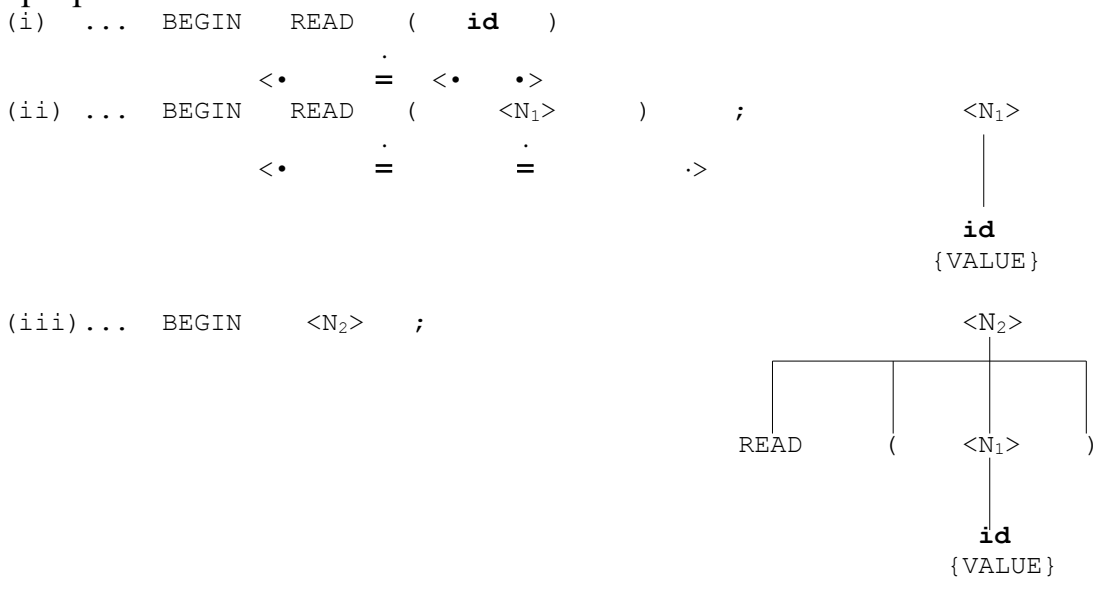
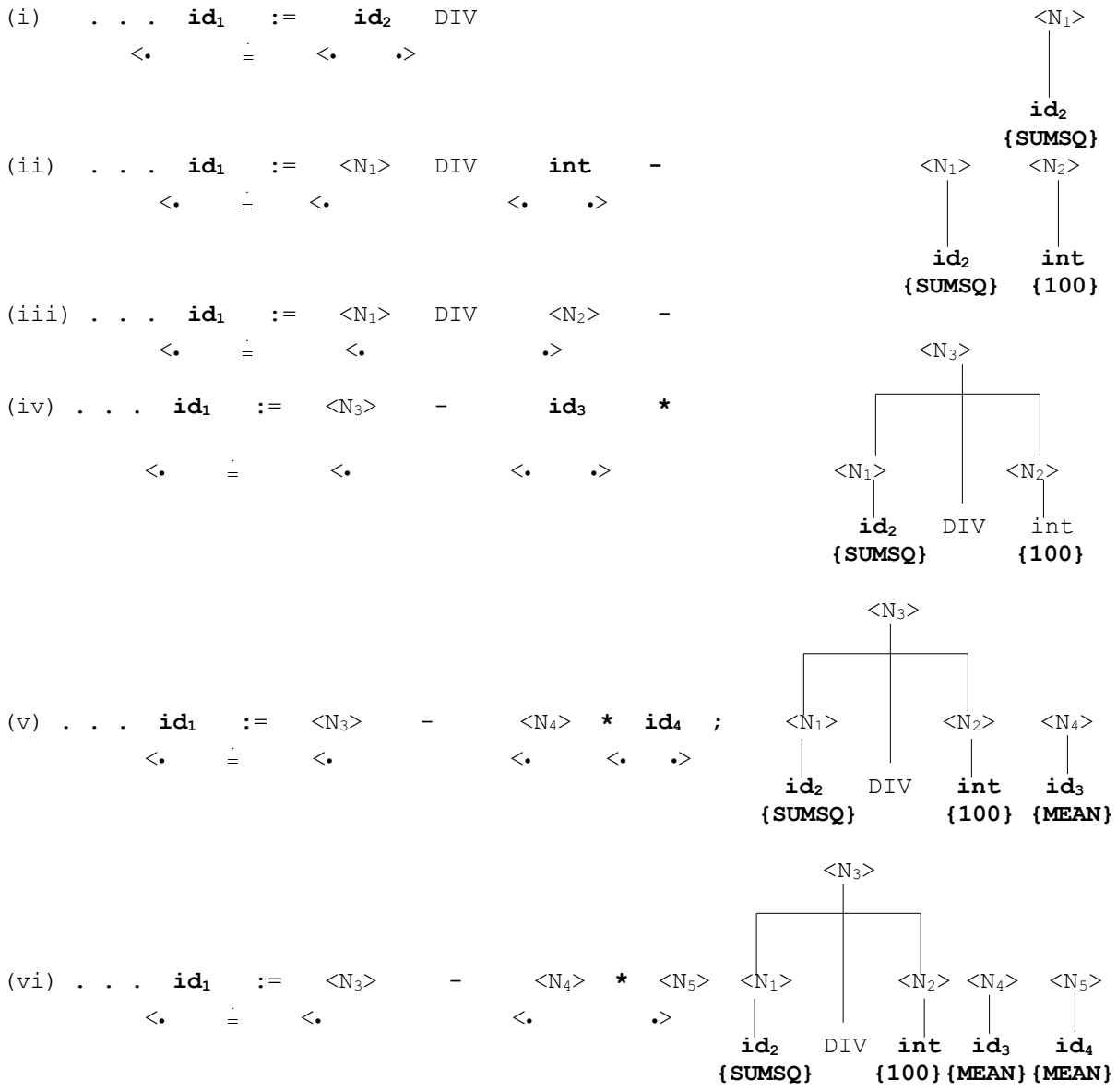


Рис. 7.9 (а)

На рис. 7.9 (б) изображен аналогичный пошаговый процесс грамматического разбора предложения присваивания в строке 14 программы на рис. 7.2. Процесс сканирования слева направо продолжается на каждом шаге грамматического разбора лишь до тех пор, пока не определился очередной фрагмент предложения для грамматического распознавания, т.е. первый фрагмент, ограниченный отношениями <• и •>. Как только подобный фрагмент выделен, он интерпретируется как некоторый очередной нетерминальный символ в соответствии с каким-нибудь правилом грамматики. Этот процесс продолжается до тех пор, пока предложение не будет распознано целиком. Каждый фрагмент дерева грамматического разбора строится, начиная с оконечных узлов вверх, в строну от корня дерева (отсюда и возник термин – восходящий разбор).



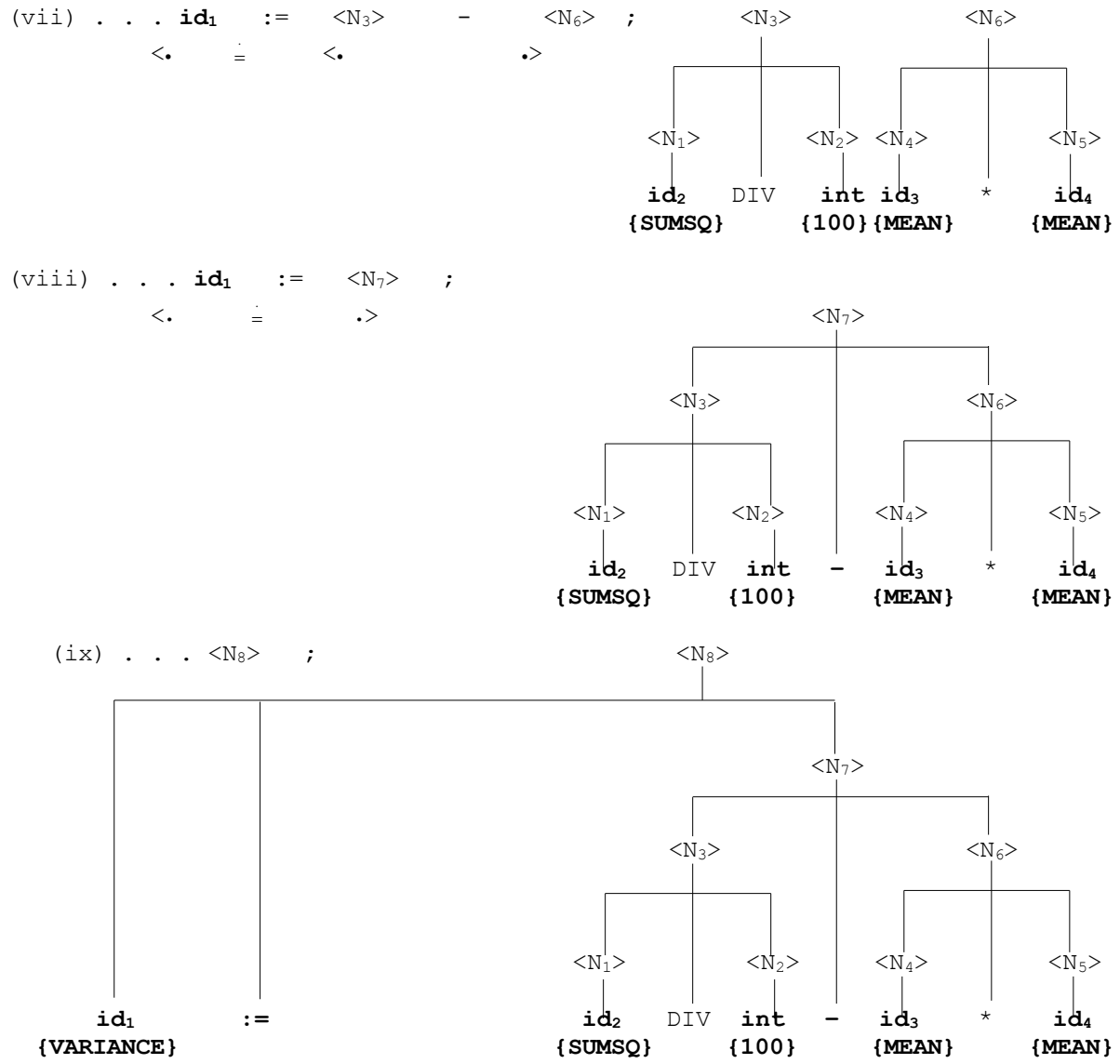


Рис.7.9 (б)

Рис. 7.9. Грамматический разбор двух предложений программы на рис. 7.2 методом операторного предшествования

Деревья грамматического разбора, изображенные на рис. 7.9 (б) и рис. 7.4 (б), имеют некоторые различия. Например, идентификатор SUMSQ на рис. 7.4 был сначала интерпретирован как <factor>, а потом как <term>, являющийся одним из операндов операции DIV. На рис. 7.9 (б) идентификатор SUMSQ интерпретирован как единственный нетерминальный символ $<N_1>$, являющийся одним из операндов операции DIV. Таким образом, $<N_1>$ на дереве, изображенном на рис. 7.9 (б), соответствует двум нетерминальным символам – <factor> и <term> – на рис. 7.4 (б). Имеются и другие подобные различия между этими двумя деревьями. Они вытекают из свободы образования имён нетерминальных символов, распознаваемых в рамках метода операторного предшествования. Интерпретация SUMSQ на рис. 7.4 (б) сначала в качестве <factor>, а потом <term> является просто переименованием

нетерминальных символов. Такое переименование необходимо, поскольку в соответствии с грамматическим правилом 11 первым операндом операции умножения должен быть <term>, а не <factor>. Так как для метода операторного предшествования имена нетерминальных символов несущественны, то подобное переименование в процессе распознавания становится ненужным. Собственно говоря, три различных имени – <exp>, <term> и <factor> – были включены в грамматику только как средства описания отношения предшествования между операторами (например, для указания того, что умножение следует выполнять прежде сложения). Поскольку эта информация содержится в матрице предшествования, то становится ненужным различать эти три имени в процессе грамматического разбора.

Метод операторного предшествования, рассмотренный для отдельных предложений, применим и для программы на рис. 7.2 в целом. При этом можно использовать матрицу предшествования на рис. 7.8.

Другой метод грамматического разбора – нисходящий метод, называемый *рекурсивным спуском*. Процессор грамматического разбора, основанный на этом методе, состоит из отдельных процедур для каждого нетерминального символа, определённого в грамматике. Каждая такая процедура старается во входном потоке найти подстроку, начинающуюся с текущей лексемы, которая может быть интерпретирована как нетерминальный символ, связанный с данной процедурой. В процессе своей работы она может вызывать другие подобные процедуры или даже рекурсивно саму себя для поиска других нетерминальных символов. Если эта процедура находит соответствующий нетерминальный символ, то она заканчивает свою работу, передаёт в вызвавшую её программу признак успешного завершения и устанавливает указатель текущей лексемы на первую лексему после распознанной подстроки. Если же процедура не может найти подстроку, которая могла бы быть интерпретирована как требуемый нетерминальный символ, она заканчивается с признаком неудачи или же вызывает процедуру выдачи диагностического сообщения и процедуру восстановления.

Пример. Грамматическое правило 13 на рис. 7.3.

Процедура метода рекурсивного спуска, соответствующая нетерминальному символу <read>, прежде всего исследует две последовательные лексемы, сравнивая их с READ и (. При совпадении эта процедура вызывает другую процедуру, соответствующую нетерминальному символу <id-list>. Если процедура <id-list> завершится успешно, то процедура <read> завершается с признаком успеха и устанавливает указатель текущей лексемы на лексему, следующую за), иначе завершается с признаком неудачи.

Эта процедура лишь немного сложнее, чем те несколько определённых грамматикой альтернатив для соответствующих нетерминалов. Дополнительно процедура должна решать, какую альтернативу попробовать в качестве следующей. Для метода

рекурсивного спуска необходимо, чтобы соответствующую альтернативу можно было выбрать на основании анализа очередной входной лексемы.

Например, процедура, соответствующая <stmt>, анализирует очередную лексему для того, чтобы выбрать одну из четырёх возможных альтернатив. Если это лексема READ, то вызывается процедура, соответствующая нетерминальному символу <read>. Если это лексема **id**, то процедура, соответствующая нетерминалу <assign>, так как это единственная альтернатива, которая может начинаться с лексемы **id** и т.д.

Попытка написать полный набор процедур для грамматики на рис. 7.3 обернётся трудностями. Процедура для <id-list>, соответствующая правилу 6, будет не в состоянии выбрать одну из двух альтернатив, поскольку обе альтернативы **id** и <id-list> могут начинаться с лексемы **id**. Тут скрыта и более существенная трудность. Если процедура каким-либо образом решит попробовать вторую альтернативу (<id-list>, **id**), то она немедленно вызовет рекурсивно саму себя для поиска нетерминального символа <id-list>. Это приведёт к ещё одному рекурсивному вызову и т.д., в результате чего образуется бесконечная цепочка рекурсивных вызовов. Причина этого заключается в том, что одна из альтернатив для <id-list> начинается также с <id-list>. Нисходящий грамматический разбор не применим непосредственно для грамматик, содержащих подобные *левые рекурсии*. Те же проблемы возникнут и в отношении правил 3, 7, 10 и 11.

```
1     <prog> ::= PROGRAM <prog-name>VAR<dec-list>BEGIN <stmt-list>END.
2     <prog-name> ::= id
3a    <dec-list>  ::= <dec> { ; <dec> }
4     <dec>       ::= <id-list> ; <type>
5     <type>      ::= INTEGER
6a    <id-list>   ::= id {, id }
7a    <stmt-list> ::= <stmt> { ; <stmt> }
8     <stmt>      ::= <assign> | <read> | <write> | <for>
9     <assign>    ::= id := <exp>
10a   <exp>       ::= <term> { + <term> | - <term> }
11a   <term>      ::= <factor> { * <factor> | DIV <factor> }
12    <factor>    ::= id | int | ( <exp> )
13    <read>      ::= READ ( <id-list> )
14    <write>     ::= WRITE ( <id-list> )
15    <for>       ::= FOR <index-exp> DO <body>
16    <index-exp> ::= id := <exp> TO <exp>
17    <body>      ::= <stmt> | BEGIN <stmt-list> END
```

Рис. 7.10 – Упрощённая грамматика Паскаля, модифицированная для грамматического разбора методом рекурсивного спуска

На рис. 7.10 изображена та же грамматика, что и на рис. 7.3, но с исключённой левой рекурсией. Для правила 6(а) на рис. 7.10:

<id-list> ::= **id** {, **id** }

Эта нотация, являющаяся широко принятым расширением БНФ, означает, что конструкция, заключённая в фигурные скобки, может быть либо опущена, либо повторяться один или более число раз. Таким образом, правило 6(а) определяет нетерминальный символ <id-list> как состоящий из

единственной лексемы **id** или же из произвольного числа следующих друг за другом лексем **id**, разделённых запятой. Это эквивалентно правилу 6 на рис. 7.3. В соответствии с этим новым определением процедура, соответствующая нетерминальному символу <id-list>, сначала ищет лексему **id**, а затем продолжает сканировать текст, пока следующая пара лексем не совпадёт с запятой и **id**. Такая запись устраняет проблему левой рекурсии, а также решает вопрос, какую из возможных альтернатив <id-list> пробовать первой.

Аналогичные изменения сделаны в правилах 3(a), 7(a), 10(a) и 11(a) на рис. 7.10. Грамматика осталась рекурсивной: <exp> определено в терминах <term>, который в свою очередь определён в терминах <factor>, а одна из альтернатив для нетерминального символа <factor> включает в себя <exp>. Это означает, что рекурсивные вызовы процедур грамматического разбора по-прежнему возможны. Однако непосредственно левая рекурсия устранена. Цепочка вызовов, начиная с <exp> и далее процедур, связанных с <term>, <factor> и опять <exp>, должна продвинуть указатель текущей лексемы из входного файла как минимум на одну лексему вперёд.

На рис. 7.11 изображен грамматический разбор методом рекурсивного спуска предложения READ в строке 9 на рис. 7.2 с использованием грамматики на рис. 7.10. На рис. 7.11(a) изображены процедуры, соответствующие нетерминальным символам <read> и <id-list>. Предполагается, что переменная TOKEN содержит тип следующей лексемы из входного потока (в рамках схемы кодирования на рис. 7.6).

В процедуре IDLIST символ запятая (,), за которым не следует лексема **id**, рассматривается как ошибка и процедура заканчивается с признаком неудачи. Если же последовательность лексем типа «**id**, **id**» является правильной конструкцией языка, то метод рекурсивного спуска не даст правильных результатов. Для такой грамматики необходимо использовать более сложные методы грамматического разбора, которые должны допускать во время разбора возврат по входной строке после обнаружения того факта, что за последней запятой не следует лексема **id**.

На рис. 7.11(б) графически представлен процесс грамматического разбора методом рекурсивного спуска для анализируемого предложения.

На фрагменте (i) изображен вызов процедуры READ, которая обнаружила лексемы READ и (во входном потоке.

Во фрагменте (ii) процедура READ вызвала процедуру IDLIST (изображено штриховой линией), которая обработала лексему **id**.

Во фрагменте (iii) процедура IDLIST закончила свою работу, передала управление процедуре READ с признаком успешного завершения; процедура READ обработала входную лексему). На этом анализ исходного предложения завершен. Процедура READ закончит свою работу с признаком успешного завершения, что означает, что нетерминальный символ <read> обнаружен. Последовательность вызова процедур и обработки лексем целиком определяется структурой предложения READ. Фрагмент (iii) полностью совпадает с деревом грамматического разбора на рис. 7.4(a). Дерево грамматического разбора строилось, начиная со своего корня, что и повлекло за собой термин *нисходящий разбор*.

```
procedure READ
   begin
      FOUND := FALSE
         if TOKEN = 8 {READ} then
            begin
               перейти к следующей лексеме
                  if TOKEN = 20 {( } then
                     begin
                        перейти к следующей лексеме
                           if IDLIST закончилась успешно
                              if TOKEN = 21 { ) } then
                                 begin
                                    FOUND := TRUE
                                      перейти к следующей лексеме
                                 end { if ) }
                     end { if ( }
            end { if READ }
          if FOUND = TRUE then
            успешное завершение
          else
            неудачное завершение
   end { READ }
procedure IDLIST
   begin
      FOUND := FALSE
         if TOKEN = 22 { id } then
            begin
               FOUND := TRUE
               перейти к следующей лексеме
               while ( TOKEN = 14 {, } ) and ( FOUND = TRUE )do
                     begin
                        перейти к следующей лексеме
                        if TOKEN = 22 { id } then
                           перейти к следующей лексеме
                        else
                           FOUND := FALSE
                     end { while }
            end { if id }
          if FOUND = TRUE then
            успешное завершение
          else
            неудачное завершение
         end { IDLIST }
```

Рис.7.11 (а)

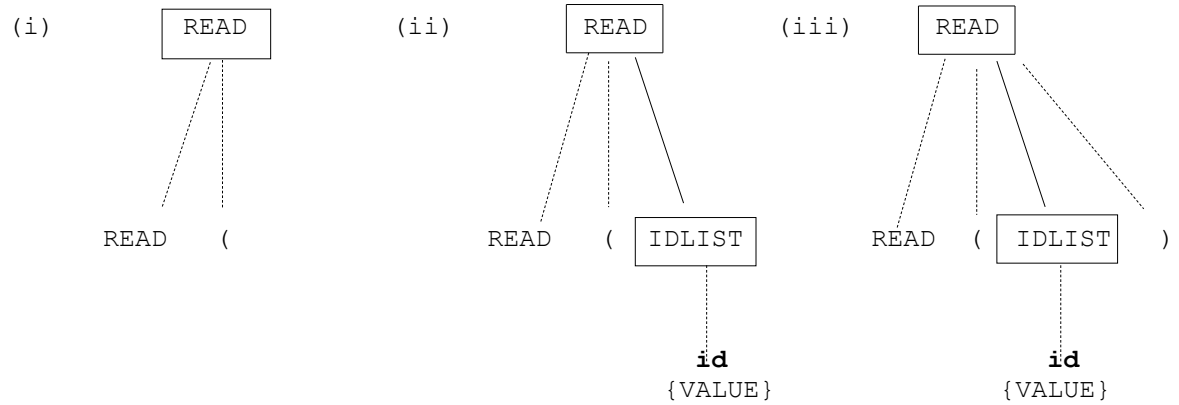


Рис. 7.11 (б)

Рис. 7.11 – Грамматический разбор предложения READ методом рекурсивного спуска

На рис. 7.12 представлен разбор методом рекурсивного спуска оператора присваивания в строке 14 на рис. 7.2. На рис. 7.12 (а) изображены процедуры для нетерминальных символов, необходимые для разбора этого предложения. На рис. 7.12(б) изображены шаги грамматического разбора (вызовы процедур и обработка лексем), аналогичные приведенным на рис. 7.11 (б). Различия между деревьями на рис. 7.12 (б) и рис. 7.4 (б) в точности соответствуют различиям между грамматиками, изображенными на рис. 7.10 и рис. 7.3.

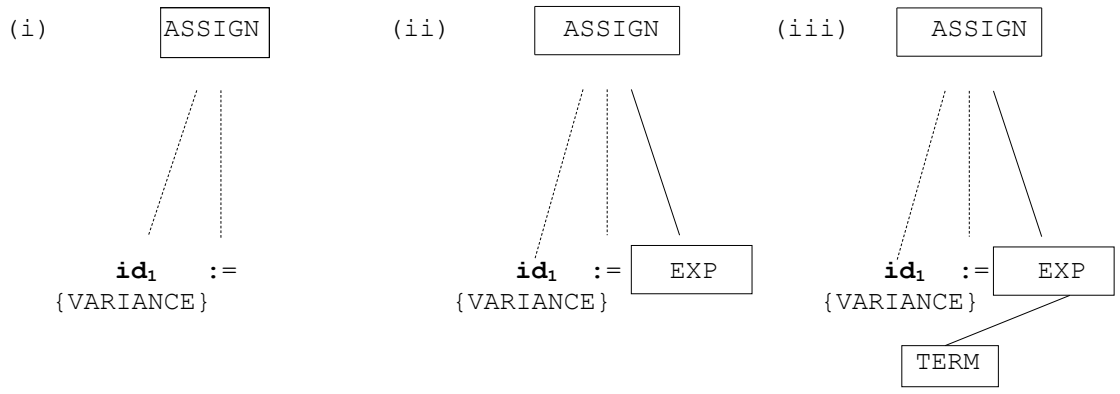
Метод рекурсивного спуска, рассмотренный на примерах грамматического разбора отдельных предложений, применим и ко всей программе в целом. В этом случае для осуществления синтаксического анализа следует просто обратиться к процедуре, соответствующей нетерминальному символу <prog>. В результате работы этой процедуры будет построено дерево грамматического разбора для всей программы.

Для разбора одной и той же программы можно использовать как отдельно восходящий (или нисходящий) метод, так и одновременно оба метода. Некоторые компиляторы используют метод рекурсивного спуска для распознавания конструкций относительно высокого уровня (например, до уровня отдельных предложений языка), а потом переключаются на метод, аналогичный методу операторного предшествования для анализа таких конструкций, как, например, арифметические выражения.

```
procedure ASSIGN
   begin
      FOUND := FALSE
      if TOKEN = 22 { id } then
            begin
               перейти к следующей лексеме
               if TOKEN = 15 { := } then
                     begin
                        перейти к следующей лексеме
                        if EXP завершилась успешно then
                           FOUND := TRUE
                     end { if := }
            end { if id }
          if FOUND = TRUE then
            успешное завершение
          else
            неудачное завершение
   end { ASSIGN }

procedure EXP
   begin
      FOUND := FALSE
      if TERM завершилась успешно then
         begin
            FOUND := TRUE
            while (( TOKEN = 16 { + } ) или ( TOKEN = 17 { - } ))
              and ( FOUND = TRUE )do
                 begin
                    перейти к следующей лексеме
                    if TERM завершилась неудачно then
                        FOUND := FALSE
                 end { while }
         end { if TERM }
      if FOUND = TRUE then
         успешное завершение
```

```
      else
         неудачное завершение
      end { EXP }

procedure TERM
   begin
      FOUND := FALSE
      if FACTOR завершилась успешно then
         begin
            FOUND := TRUE
            while (( TOKEN = 18 { * } ) или (TOKEN = 19 { DIV } ))
              and ( FOUND = TRUE )do
                 begin
                    перейти к следующей лексеме
                    if FACTOR завершилась неудачно then
                        FOUND := FALSE
                 end { while }
         end { if FACTOR }
      if FOUND = TRUE then
         успешное завершение
      else
         неудачное завершение
      end { TERM }

procedure FACTOR
   begin
      FOUND := FALSE
      if (( TOKEN = 22 { id } ) или ( TOKEN = 23 { int } )) then
         begin
            FOUND := TRUE
            перейти к следующей лексеме
         end  { if id или int}
      else
         if TOKEN = 20 { ( } then
            begin
               перейти к следующей лексеме
               if EXP завершилась успешно then
               if TOKEN = 21 { ) } then
                  begin
                     FOUND := TRUE
                     перейти к следующей лексеме
                  end { if ) }
            end { if ( }
          if FOUND = TRUE then
            успешное завершение
          else
            неудачное завершение
   end { FACTOR }
```

Рис. 7.12 (a)



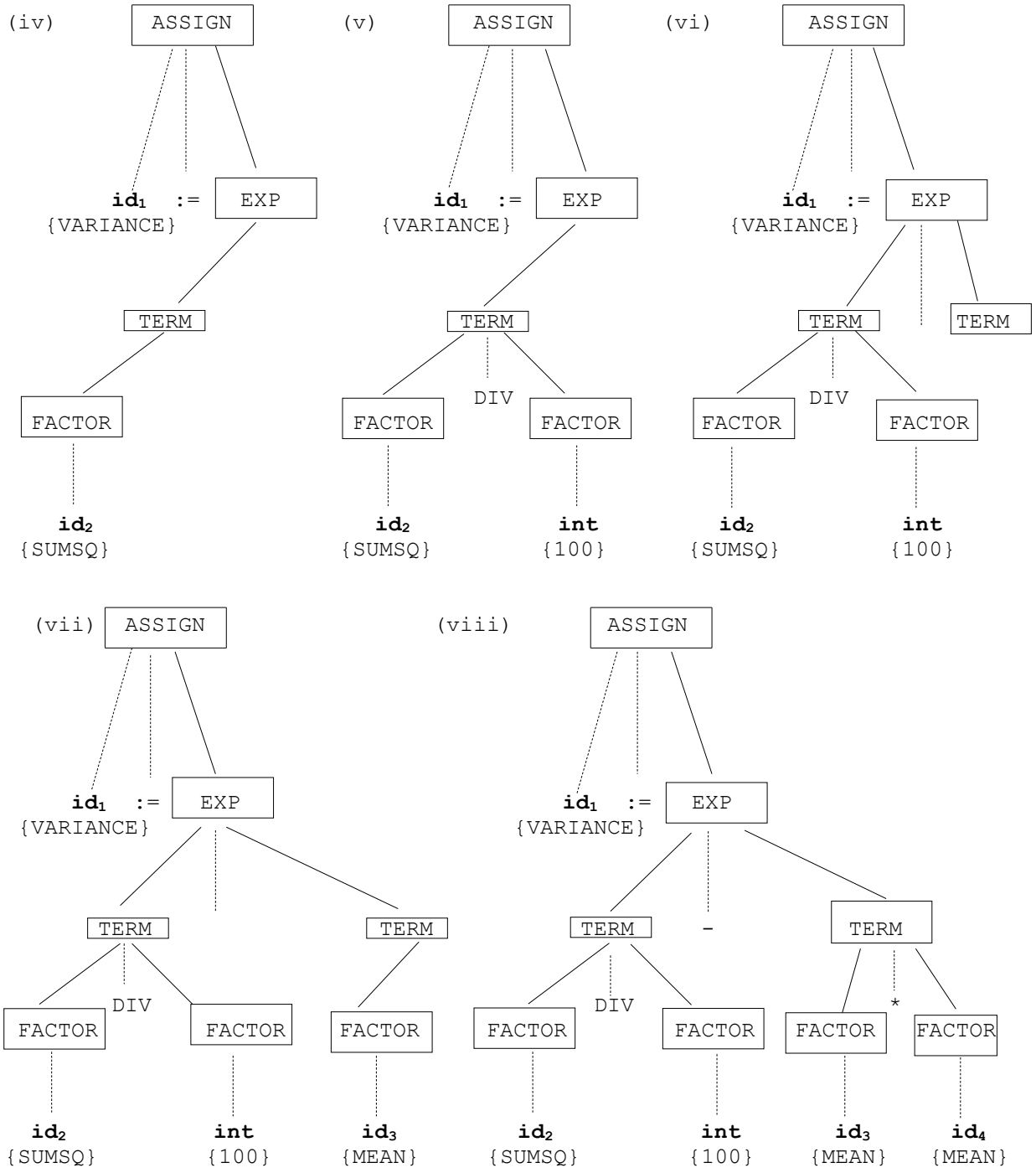


Рис. 7.12 (б)

Рис. 7.12 – Грамматический разбор предложения присваивания методом рекурсивного спуска

2. Контрольные вопросы по теме

1. Опишите общую схему компиляции.

2. Опишите понятия: «грамматика языка программирования», «нетерминальные символы», «терминальные символы». Что такое неоднозначная грамматика?

3. Что включает в себя понятие «лексический анализ»?
4. Охарактеризуйте работу сканера.
5. Опишите два класса методов грамматического разбора.
6. Метод операторного предшествования.
7. Метод рекурсивного спуска.

3. Варианты индивидуальных заданий

Составить программу лексического и синтаксического анализа тестовой программы, содержащей директивы определения данных DB и DW, а также команды, согласно варианту.

ВАРИАНТ 1. Команды тестовой программы: MOV, IMUL, IDIV.
ВАРИАНТ 2. Команды тестовой программы: ADD, IDIV, POP.
ВАРИАНТ 3. Команды тестовой программы: POP, IMUL, SUB.
ВАРИАНТ 4. Команды тестовой программы: MOV, POP, IMUL.
ВАРИАНТ 5. Команды тестовой программы: ADD, PUSH, SUB.
ВАРИАНТ 6. Команды тестовой программы: SUB, IMUL, POP.
ВАРИАНТ 7. Команды тестовой программы: POP, IDIV, MOV.
ВАРИАНТ 8. Команды тестовой программы: SUB, PUSH, IMUL.
ВАРИАНТ 9. Команды тестовой программы: MOV, IMUL, PUSH.
ВАРИАНТ 10. Команды тестовой программы: ADD, PUSH, POP.
ВАРИАНТ 11. Команды тестовой программы: PUSH, IMUL, MOV.
ВАРИАНТ 12. Команды тестовой программы: SUB, POP, ADD.
ВАРИАНТ 13. Команды тестовой программы: MOV, PUSH, IDIV.
ВАРИАНТ 14. Команды тестовой программы: ADD, IMUL, POP.
ВАРИАНТ 15. Команды тестовой программы: IMUL, IDIV, ADD.

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

ЛАБОРАТОРНАЯ РАБОТА №8
ТЕМА: ГЕНЕРАЦИЯ ОБЪЕКТНОГО КОДА

Цель: изучение принципов работы семантических программ

1. Методические указания к выполнению работы

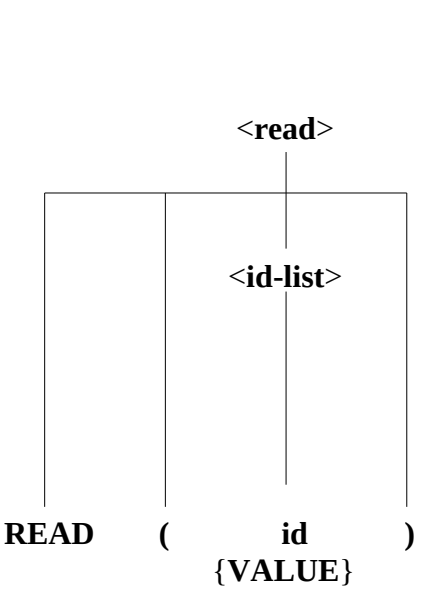
1.1. Генерация объектного кода

После анализа синтаксиса программы последним шагом процесса компиляции является генерация объектного кода. Рассмотрим метод, который генерирует объектный код для каждого фрагмента программы, как только распознан синтаксис этого фрагмента. Для реализации рассматриваемого метода генерации объектного кода необходим набор подпрограмм, соответствующих каждому правилу и каждой альтернативе в правилах грамматики. Как только в результате процесса грамматического разбора будет распознан фрагмент текста исходной программы, соответствующий некоторому правилу грамматики, вызывается подпрограмма, соответствующая этому правилу. Эти программы называют *семантическими программами*, поскольку выполняемые ими действия связаны со смыслом, связанным с соответствующими конструкциями языка. Семантические программы непосредственно генерируют объектный код. Поэтому они называются *программами генерации кода.*

Рассмотрим программы генерации кода, предназначенные для использования с грамматикой на рис. 7.3. Генерируемый код, очевидно, зависит от того, на каком компьютере будет выполняться компилируемая программа. Для своих рабочих данных рассматриваемые программы генерации кода будут использовать две структуры: список и стек. Переменная LISTCOUNT используется в качестве счетчика элементов, содержащихся в данный момент в списке. Программы генерации кода также используют спецификаторы лексем, обозначаемые S (лексема). Для лексемы **id** выражение S(**id**) является именем соответствующего идентификатора или указателем на него в таблице символов. Для лексемы **int** выражение S(**int**) является значением соответствующего целого числа, например #100. Будем считать, что после генерации каждого фрагмента объектного кода указатель свободной памяти LOCCTR модифицируется таким образом, чтобы он все время указывал на первую свободную ячейку памяти компилируемой программы.

Рис. 8.1 иллюстрирует процесс генерации объектного кода для предложения READ в строке 9 программы на рис 7.2. Дерево грамматического разбора этого предложения представлено на рис. 8.1(а). Это дерево можно получить многими различными методами грамматического разбора. На каждом шаге процесса разбора распознается самая левая подстрока входного текста, которая может быть интерпретирована в соответствии с каким-либо из правил грамматики. В методе операторного предшествования подобное распознавание осуществляется, когда подстрока

входного текста редуцируется к некоторому нетерминальному символу <N_1>. В методе рекурсивного спуска распознавание происходит в тот момент, когда соответствующая процедура заканчивает свою работу с признаком успешного завершения. Таким образом, в процессе грамматического разбора всегда будет сначала распознан идентификатор VALUE в качестве нетерминального символа <id-list>, а уже потом предложение в целом будет распознано в качестве нетерминального символа <read>.



<id-list> ::= id
добавить S(id) к списку
увеличить LISTCOUNT на 1

<id-list> ::= <id-list> , id
добавить S(id) к списку
увеличить LISTCOUNT на 1

<read>::= READ < <id-list> >

Рис. 8.1(а)

```
сгенерировать [ +JSUB XREAD]
внешние ссылки для XREAD
сгенерировать [ +WORD LISTCOUNT]
for каждый элемент списка do
  begin
   удалить S(ITEM) из списка
   сгенерировать [ WORD S(ITEM)]
  end
LISTCOUNT := 0
```

Рис .8.1(б)

```
+JSUB    XREAD
WORD     1
WORD     VALUE
```

Рис. 8.1(в)

Рис. 8.1. Генерация объектного кода для предложения READ

На рис. 8.1(в) символически изображен объектный код, который должен быть сгенерирован для предложения READ. Этот код представляет собой вызов подпрограммы XREAD, которая должна содержаться в стандартной библиотеке компилятора. Эта подпрограмма может быть

вызвана также любой программой, желающей выполнить операцию чтения. Подпрограмма XREAD связывается с генерируемой программой связывающим загрузчиком или же редактором связей. (Компилятор должен включить в объектную программу достаточно информации для определения всех необходимых связей.) Подобная техника обычно используется при компиляции предложений, выполняющих относительно сложные функции. Поскольку подпрограмма XREAD может быть использована для любых операций чтения, она должна иметь параметры, определяющие детали этой операции. В данном случае список параметров подпрограммы XREAD помещается непосредственно за инструкцией JSUB, которая ее вызывает. Первое слово в этом списке параметров содержит величину, определяющую количество переменных, которым должно быть присвоено значение в результате осуществления операции чтения. В последующих словах содержаться адреса этих переменных. Таким образом, вторая строка на рис. 8.1(в) определяет, что должно быть прочитано значение одной переменной, а третья строка содержит адрес. Адрес первого слова списка параметров будет автоматически помещен в регистр L при выполнении инструкции JSUB. Подпрограмма XREAD может использовать этот адрес для определения своих параметров и, сложив содержимое регистра L с длиной списка параметров, определить значение адреса возврата.

На рис. 8.1 (б) представлен набор программ, которые могут использоваться для генерации объектного кода. Первые две программы соответствуют двум альтернативам для нетерминального символа <id-list> в грамматическом правиле 6 на рис. 7.3. В любом случае спецификатор лексемы S(**id**) для очередного идентификатора, являющегося элементом <id-list>, включается в список, используемый программами генерации кода. Соответствующая переменная LISTCOUNT увеличивается на единицу, чтобы отразить включение в список нового идентификатора. После того как нетерминальный символ <id-list> будет разобран, список будет содержать спецификаторы лексем для всех идентификаторов, содержащихся в < id-list >. Когда будет распознано предложение <read>, эти спецификаторы лексем будут удалены из списка и использованы для генерации объектного кода, соответствующего операции чтения READ.

Следует обратить внимание, что в процессе генерации дерева грамматического разбора, изображенного на рис. 8.1(а), вначале распознается <id-list>, а потом уже <read>. На каждом шаге грамматического разбора вызывается соответствующая программа генерации объектного кода.

На рис. 8.2 изображен процесс генерации объектного кода для предложения присваивания в строке 14 на рис 7.2.

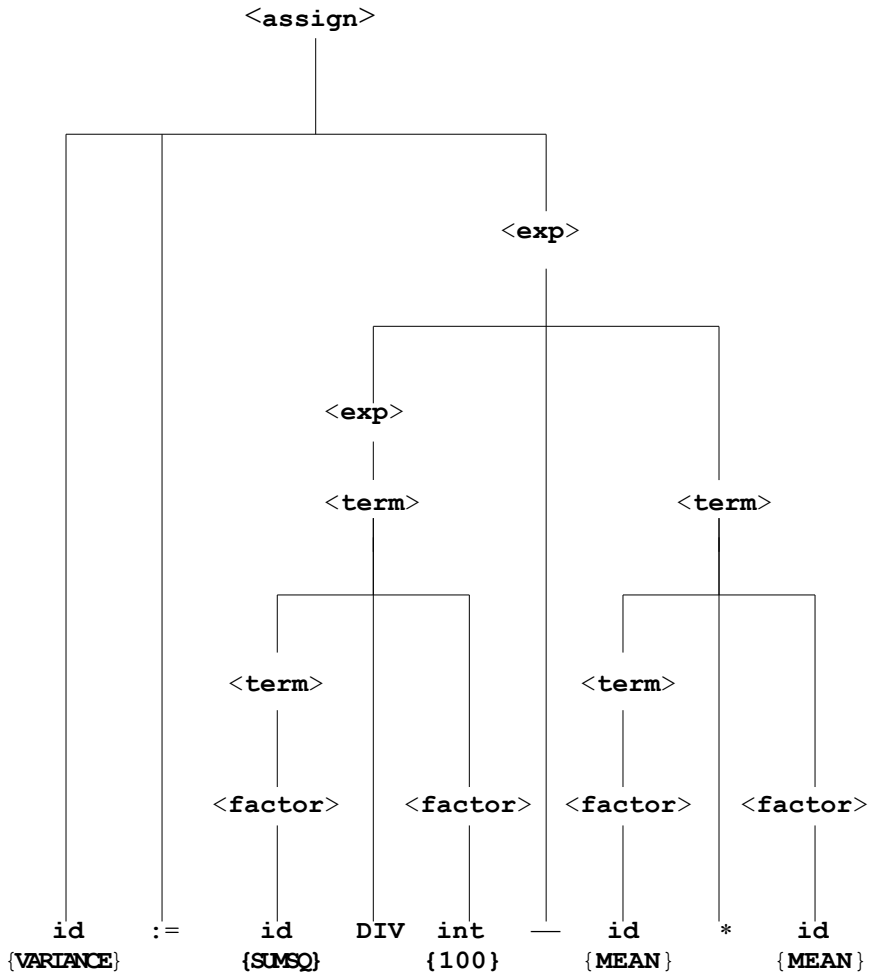


Рис. 8.2 (а)

```
<assign> ::= id := <exp>
                GETA (<exp>)
                сгенерировать [   STA   S(id)]
                REGA := null

<exp> ::= <term>
                S(<exp>) := S(<term>)
                if S(<exp>) = rA then
                   REGA := <exp>

<exp>1 ::= <exp>2 + <term>
                if S(<exp>2) = rA then
                   сгенерировать [   ADD   S(<term>)]
                else if S(<term>) = rA then
                   сгенерировать [   ADD   S(<exp>2)]
                else
                   begin
                      GETA (<exp>2)
                      сгенерировать [   ADD   S(<term>)]
                   end
                S(<exp>1) := rA
                REGA := <exp>1
<exp>1::=<exp>2 - <term>
                if S(<exp>2) = rA then
                   сгенерировать [   SUB   S(<term>)]
                else
                   begin
```

```
                        GETA (<exp>2)
                        сгенерировать [   SUB   S(<term>)]
                     end
                  S(<exp>1) := rA
                  REGA :=<exp>1

<term> ::= <factor>
                  S(<term>):= S(<factor>)
                  if S(<term>) = rA then
                     REGA := <term>

<term>1::= <term>2 * <factor>
                  if S(<term>2) = rA then
                     сгенерировать [   MUL   S(<factor>)]
                  else if S(<factor>) = rA then
                     сгенерировать [   MUL   S(<term>2)]
                  else
                     begin
                        GETA (<term>2)
                        сгенерировать [   MUL   S(<factor>)]
                     end
                  S(<term>1) := rA
                  REGA := <term>1

<term>1 ::= <term>2 DIV <factor>
                  if S(<term>2) = rA then
                     сгенерировать [   DIV   S(<factor>)]
                  else
                     begin
                        GETA (<term>2)
                        сгенерировать [   DIV   S(<factor>)]
                     end
                  S(<term>1) := rA
                  REGA := <term>1

<factor> ::= id
                  S(<factor>):= S(id)

<factor>::= int
                  S(<factor>):= S(int)

<factor> ::= ( <exp>)
                  S(<factor>) := S(<exp>)
```

Рис. 8.2(б)

```
procedure GETA (NODE)
   begin
       if REGA = null then
          сгенерировать [   LDA   S(NODE)]
          else if S(NODE) = rA then
             begin
                породить новые рабочие переменные Ti
                сгенерировать [   STA   Ti]
                ссылки вперед на Ti
                S(REGA) := Ti
                сгенерировать [   LDA   S(NODE)]
             end {if rA}
          S(NODE) := rA
          REGA := NODE
   end {GETA}
```

Рис. 8.2 (в)

```
LDA   SUMSQ
DIV   #100
STA   T1
LDA   MEAN
MUL   MEAN
STA   T2
LDA   T1
SUB   T2
STA   VARIANCE
```

Рис .8.2 (г)

Рис. 8.2 – Генерация объектного кода для предложения присваивания

На рис. 8.2(а) приведено дерево грамматического разбора этого предложения. Основная работа при разборе предложения состоит в анализе нетерминального символа <exp> в правой части оператора присваивания. В процессе грамматического разбора идентификатор SUMSQ распознается сначала как <factor>, потом как <term>. Далее распознается целое число 100 как <factor>. Затем фрагмент SUMSQ DIV 100 распознается как <term> и т.д. Цепочка шагов примерно совпадает с рис 7.9(б). Порядок распознавания фрагментов этого предложения совпадает с порядком, в котором должны выполняться соответствующие вычисления. Сначала вычисляются подвыражения SUMSQ DIV 100 и MEAN * MEAN, а затем второй результат вычитается из первого.

Как только очередной фрагмент предложения распознан, вызывается соответствующая программа генерации кода.

Пример. Сгенерировать код, соответствующий правилу

<term>$_1$ ::= <term>$_2$<factor>*

Индексы здесь использованы для различения двух вхождений нетерминального символа <term>. Данные программы генерации кода выполняют все арифметические операции с использованием регистра A, и следует заведомо сгенерировать в объектном коде операцию MUL. Результат этого умножения <term>$_1$ после операции MUL сохранится в регистре A. Если либо <term>$_2$, либо <factor> уже находятся в регистре A (после выполнения предыдущих вычислений), генерация инструкции MUL – это все, что требуется. Иначе нужно сгенерировать также инструкцию LDA, предшествующую инструкции MUL. Также надо предварительно сохранить значение регистра A, если оно понадобится в будущем.

Очевидно, необходимо отслеживать значение, помещенное в регистр A, после каждого фрагмента генерируемого объектного кода. Это можно осуществить за счет расширения понятия спецификатора лексемы на нетерминальные узлы дерева грамматического разбора. В рассмотренном примере *спецификатору узла* S(<term>$_1$) будет присвоено значение rA, указывающее на то, что результат вычислений содержится в регистре A. Переменная REGA используется для указания на самый высокий уровень узла дерева грамматического разбора, значение которого помещено в

регистр A в сгенерированном до данного момента объектном коде (т.е. указатель на узел, спецификатор которого равен rA). В каждый момент процесса генерации объектного кода существует ровно один такой узел. Если же в регистре A нет значения, соответствующего некоторому узлу, то спецификатор этого узла аналогичен спецификатору лексемы: это либо указатель на переменную (в таблице символов), содержащую соответствующее значение, либо указатель на целую константу.

Рассмотрим программу генерации кода на рис. 8.2 (б), соответствующую правилу <*term*>$_1$ *::=* <*term*>$_2$*<*factor*>

Если спецификатор узла какого-либо из операндов равен rA, то соответствующее значение уже содержится в регистре A, и программа генерирует только одну инструкцию MUL. Адрес операнда для инструкции MUL содержится в спецификаторе узла другого операнда (значение которого не находится на регистре A). Иначе вызывается процедура GETA, изображенная на рис. 8.2 (в). Она генерирует инструкцию LDA для загрузки значения, связанного со значением <term>$_2$, в регистр A. Дополнительно перед инструкцией LDA процедура генерирует инструкцию STA для сохранения текущего значения регистра A, если только REGA не нуль (равенство REGA нулю означает, что это значение больше не понадобится). Это значение запоминается в некоторой *рабочей* переменной. Рабочие переменные образуются в процессе генерации объектного кода (с именами T1, T2, …) по мере необходимости. Для рабочих переменных будет отведено место в конце объектной программы. На узел дерева грамматического разбора, связанный со значением, содержащимся в регистре A, указывает переменная REGA. Спецификатор этого узла модифицируется, чтобы указывать на рабочую переменную, используемую для хранения этого значения.

После того как все необходимые инструкции сгенерированы, программа генерации кода устанавливает спецификатор S (<term>$_1$) и переменную REGA таким образом, чтобы было видно, что значение, соответствующее <term>$_1$, находится в настоящее время в регистре A. На этом процедура генерации кода для операции * завершается.

Программа генерации кода, соответствующая операции +, почти совпадает с рассмотренной программой для операции *. Программы для DIV и «–» также аналогичны, за исключением того, что для этих операций необходимо, чтобы на регистре A был установлен именно первый операнд. Программа генерации кода для операции присваивания <assign> состоит в загрузке присваиваемой величины на регистр A (с использованием GETA) и генерации инструкции STA. Переменная REGA потом обнуляется, так как код, соответствующий предложению присваивания, полностью сгенерирован и промежуточные результаты больше не нужны.

Оставшиеся правила, показанные на рис. 8.2 (б), не требуют генерации каких-либо машинных инструкций, поскольку они не соответствуют никаким вычислениям или перемещениям данных.

Программы генерации кодов для этих правил должны соответствующим образом установить спецификатор для узла самого верхнего уровня, чтобы он указывал на ячейку, содержащую соответствующее значение.

На рис. 8.2 (б) представлено символическое изображение объектного кода, сгенерированного для предложения присваивания.

На рис. 8.3 изображены другие программы генерации кода для грамматики на рис. 7.3.

```
<prog> ::= PROGRAM <prog-name> VAR <dec-list> BEGIN <stmt-list> END
                сгенерировать [LDL   RETADR]
                сгенерировать [RSUB]
                for каждая используемая переменная Ti do
                  сгенерировать [Ti   RESW   1]
                вставить [J   EXADDR] {переход на
                        первое выполняемое предложение} в
                        байты 3-5 объектной программы
                сформировать ссылки вперед на переменные Ti
                сгенерировать запись-модификатор для внешних ссылок
                сгенерировать [END]

<prog-name> ::= id
                сгенерировать [START   0]
                сгенерировать [EXTREF  XREAD,XWRITE]
                сгенерировать [STL     RETADR]
                увеличить LOCCTR на 3 {место для команды
                        перехода на первое выполняемое предложение}
                сгенерировать [RETADR  RESW   1]

<dec-list> ::= {любые альтернативы}
                сохранить LOCCTR в качестве EXADDR {адрес
                        первого выполняемого предложения}

<dec> ::= <id-list> : <type>
                for каждый элемент списка do
                   begin
                      исключить S(NAME) из списка
                      занести LOCCTR в таблицу символов в качестве
                                 адреса NAME
                      сгенерировать [S(NAME)   RESW   1]
                   end
                LISTCOUNT := 0

<type> ::= INTEGER
                {не нужны никакие действия по генерации кода}

<stmt-list> ::= {любые альтернативы}
                {не нужны никакие действия по генерации кода}

<stmt> ::= {любые альтернативы}
                {не нужны никакие действия по генерации кода}

<write> ::= WRITE(<id-list>)
                сгенерировать [+JSUB XWRITE]
                внешние ссылки для XWRITE
                сгенерировать [WORD   LISTCOUNT]
                for каждый элемент списка do
                  begin
                     исключить S(ITEM) из списка
                     сгенерировать [WORD   S(ITEM)]
                  end
                LISTCOUNT := 0
```

```
<for> ::= FOR <index-exp> DO <body>
      взять из стека JUMPADDR {адрес команды выхода из цикла}
      взять из стека S(INDEX) {индексная переменная}
      взять из стека LOOPADDR {адрес начала цикла}
      сгенерировать [LDA   S(INDEX)]
      сгенерировать [ADD   *1]
      сгенерировать [J     LOOPADDR]
  вставить [JGT LOCCTR] в ячейку JUMPADDR

<index-exp> ::=id := <exp>1 TO <exp>2
                GETA (<exp>1)
                поместить в стек LOCCTR {адрес начала цикла]
                поместить в стек S(id) {индексная переменная}
                сгенерировать [STA   S(id)]
                сгенерировать [COMP   S(<exp>2)]
                поместить в стек LOCCTR {адрес выхода из цикла}
                увеличить LOCCTR на 3 {место для команды перехода}
                REGA := null

<body> ::= {любые альтернативы}
                {не нужны никакие действия по генерации кода}
```

Рис. 8.3 – Другие программы генерации кода для грамматики на рис. 7.3

Программа для <prog-name> генерирует заголовок объектной программы и инструкции для сохранения адреса возврата и перехода на первую выполняемую инструкцию компилируемой программы. Когда вся программа распознана, резервируется память для рабочих переменных (T_i). Затем все ссылки на эти переменные фиксируются в объектном коде с использованием процедуры обработки ссылок вперед. Компилятор генерирует также модифицирующие записи, необходимые для описания внешних ссылок на библиотечные подпрограммы.

После того как распознан нетерминальный символ <index-exp>, генерируется код для инициализации индексной переменной цикла и инструкции проверки конца цикла. Часть информации также записывается в стек для дальнейшего использования. Далее независимо генерируются коды для каждого предложения, составляющего тело цикла. После того как вся конструкция <for> распознана, генерируется код, увеличивающий индексную переменную и осуществляющий возврат на начало цикла для проверки условий его завершения. Здесь программы генерации кода используют информацию, записанную в стек программой, соответствующей <index-exp>.

Таблица 8.1 – Символическое представление кода для программы на рис. 7.2

Строка	**Символическое представление сгенерированного кода**			
1	STATS	START	0	{ заголовок программы }
		EXTREF	XREAD, XWRITE	
		STL	RETADR	{ сохранение адреса возврата }
		J	{EXADDR}	
	RETADR	RESW	1	

Продолжение таблицы 8.1

3	SUM	RESW	1	
	SUMSQ	RESW	1	
	I	RESW	1	
	VALUE	RESW	1	
	MEAN	RESW	1	
	VARIANCE	RESW	1	
5	{EXADDR}	RESW	#0	{SUM := 0}
		STA	SUM	
6		LDA	#0	{SUMSQ := 0}
		STA	SUMSQ	
7		LDA	#1	{FOR I:=1 TO 100}
		COMP	#100	
		JGT	{L2}	
	(L1)	STA	I	
9		+JSUB	XREAD	(READ(VALUE))
		WORD	1	
		WORD	VALUE	
10		LDA	SUM	{SUM := SUM + VALUE}
		ADD	VALUE	
		STA	SUM	
11		LDA	VALUE	{SUMSQ :=SUMSQ + VALUE * VALUE}
		MUL	VALUE	
		ADD	SUMSQ	
		STA	SUMSO	
		LDA	1	{конец цикла FOR}
		ADD	#1	
		J	{L1}	
13	{L2}	LDA	SUM	{MEAN := SUM DIV 100}
		DIV	#100	
		STA	MEAN	
14		LDA	SUMSQ	{VARIANCE := SUMSQ DIV 100 – MEAN * MEAN}
		DIV	#100	
		STA	T1	
		LDA	MEAN	
		MUL	MEAN	
		STA	T2	
		LDA	T1	
		SUB	T2	
		STA	VARIANCE	
15		+JSUB	XWRITE	{WRITE(MEAN,VARIANCE)}
		WORD	2	
		WORD	MEAN	
		WORD	VARIANCE	
		LDL	RETADR	{ возврат }
	T1	RESW	1	{ используемые рабочие переменные }
	T2	RESW	1	
		END		

2. Контрольные вопросы по теме

1. Дайте определение семантической программы.
2. От чего зависит генерируемый код?
3. Принцип обработки данных, попавших в список, стек.
4. Для чего расширяют понятие спецификатора лексемы?

5. В чем основное отличие программ генерации кода для +, * от программ генерации кода для DIV, – ?

3. Варианты индивидуальных заданий

Составить программу генерации объектного кода тестовой программы, содержащей директивы определения данных DB и DW, а также команды согласно варианту из лабораторной работы №7.

Результатом работы программы должен быть текстовый файл для тестовой программы вида:

1	2	3	4	5
Адрес	Код	Метка	Команда или директива	Операнды или данные
0000	0A	A	DB	10
0001	0700	B	DW	7
0003	50		PUSH	AX
0004	03D8		ADD	BX, AX
0006	F6E3		MUL	BL
0008	59		POP	CX
0009	. . . . .			

Колонки 3 – 5 – это код тестовой программы. Колонки 1 – 2 генерируются разработанной программой.

Объектные коды всех используемых команд приведены ниже.

Объектный код для команды IMUL (1 вариант):
|1111011w|mod101r/m|

Объектный код для команды IDIV (1 вариант):
|1111011w|mod111r/m|

Объектный код для команды PUSH (3 варианта):

1. *Регистр*: |01010reg|
2. *Сегментный регистр*: |000sg111|
3. *Память*: |11111111| mod110r/m|

Объектный код для команды POP (3 варианта):

1. *Регистр*: |01011reg|
2. *Сегментный регистр*: |000sg111|
3. *Память*: |10001111| mod000r/m|

Объектный код для команды MOV (7 вариантов)

1. *Регистр/память в/из регистр*: |100010dw|modregr/m|

2. Непосредственное значение в регистр/память:
|1100011w|mod000r/m|--data--|data, если w=1|

3. Непосредственное значение в регистр:
|1011wreg|--data--|data, если w=1|

4. *Память в регистр AX (AL)*: |1010000w|addr-low|addr-high|
5. *Регистр AX (AL) в память*: |1010001w|addr-low|addr-high|
6. *Регистр/память в сегментный регистр*: |10001110|mod0sgr/m|
7. *Сегментный регистр в регистр/память*: |10001100|mod0sgr/m|

Объектный код для команды ADD (3 варианта):

1. *Регистр плюс регистр или память*: |000000dw|modregr/m|
2. *Регистр AX (AL) плюс непосредственное значение*: |0000010w|--data--|data, если w=1|
3. *Регистр или память плюс непосредственное значение*: |100000sw|mod000r/m|--data--|data, если sw=01|

Объектный код для команды SUB (3 варианта)

1. *Регистр из регистра или памяти*: |001010dw|modregr/m|
2. *Непосредственное значение из регистра AX (AL)*: |0010110w|--data--|data, если w=1|
3. *Непосредственное значение из регистра или памяти*: |100000sw|mod101r/m|--data--|data, если sw=01|

Использованы следующие сокращения:
reg – трех-битовый указатель регистра;
sg – двух-битовый указатель сегментного регистра;
data – непосредственный операнд (8 бит при w=0 и 16 бит при w= 1);
addr-high – первый (старший) байт адреса;
addr-low – левый (младший) байт адреса;
Бит *d* – указывает направление потока между операндом 1 и операндом 2.

В таблицах 8.2 – 8.5 приведены значения битов **w**, **mod** и **r/m**.

Таблица 8.2 – Основные, базовые и индексные регистры (**reg**)

Биты	**w=0**	**w=1**
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Таблица 8.3 – Сегментные регистры (**sg**)

Биты	Сегментный регистр
00	ES
01	CS
10	SS
11	DS

Таблица 8.4. – биты **mod**, определяющие адресацию регистра или памяти

Биты **mod**	Значение
00	Биты r/m дают абсолютный адрес памяти, байт смещения отсутствует
01	Биты r/m дают абсолютный адрес памяти, и имеется один байт смещения
10	Биты r/m дают абсолютный адрес памяти, и имеются два байта смещения
11	Биты r/m определяют регистр.

Таблица 8.5 – три бита **r/m** (регистр/память), определяющие совместно с битами **mod** способ адресации

r/m	mod=00	mod=01	mod=10	mod=11	
				w=0	w=1
000	BX+SI	BX+SI+disp	BX+SI+disp	AL	AX
001	BX+DI	BX+DI+disp	BX+DI+disp	CL	CX
010	BP+SI	BP+SI+disp	BP+SI+disp	DL	DX
011	BP+DI	BP+DI+disp	BP+DI+disp	BL	BX
100	SI	SI+disp	SI+disp	AH	SP
101	DI	DI+disp	DI+disp	CH	BP
110	Direct	BP+disp	BP+disp	DH	SI
111	BX	BX+disp	BX+disp	BH	DI

Пример объектного кода для команды ADD BX, AX.

0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0
						d	w	mod		reg			r/m		

d = 1 означает, что биты reg и w описывают операнд 1 (BX), а mod, r/m и w – операнд 2 (AX);
w = 1 определяет размер регистров в одно слово;
mod = 11 указывает что операнд 2 является регистром;
reg = 011 указывает, что операнд 1 является регистром BX;
r/m = 000 указывает, что операнд 2 является регистром AX.

Пример объектного кода для команды MUL BL.

1	1	1	1	0	1	1	0	1	1	1	0	0	0	1	1
							w	mod		reg			r/m		

4. Порядок выполнения лабораторной работы

1. Изучить теоретическую часть.
2. Письменно ответить на контрольные вопросы.
3. Выполнить индивидуальное задание на компьютере.
4. Оформить отчет.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. Бек Л. Введение в системное программирование / Л. Бек. – М.: Мир, 1988. – 448 с.

2. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT / Р. Джордейн. – М.: Финансы и статистика, 1992. – 544 с.

3. Злобин В. К. Программирование арифметических операций в микропроцессорах / В. К. Злобин, В. Л. Григорьев. – М.: Высшая школа, 1991. – 303 с.

4. TECH Help! Справочная система по прерываниям и функциям BIOS, DOS и портам оборудования. Электронный справочник. Flambeaux Software.

5. Абель Питер. Язык Ассемблера для IBM PC и программирование / Питер Абель. – М.: Высшая школа, 1992. – 447 с.

6. Скенлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке Ассемблера / Л. Скенлон. – М.: Радио и связь, 1991. – 336 с.

Навчальне видання

Котенко Владислав Миколайович

Методичні вказівки
до виконання та оформлення лабораторних робіт до курсу
«Операційні системи»
(Рос. мовою)

План вид. 2014 р., поз. № 117