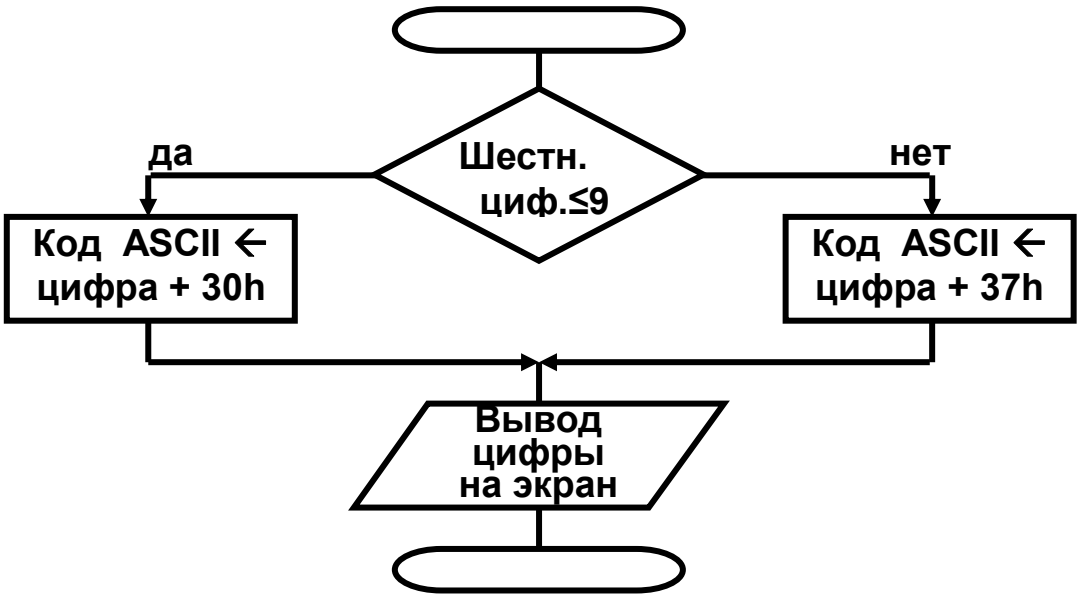
Министерство образования и науки

ЭВМ И ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА

Раздел 1
"Ассемблер для процессора i8086"

Учебное методическое пособие



Санкт-Петербург- 2017

Министерство образования и науки

Кафедра копьютерных технологий и электронного обучения

ЭВМ И ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА
Раздел 1
Ассемблер для процессора i8086
Лабораторный практикум

Учебное методическое пособие для
студентов педагогических вузов

2017

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

Цель дисциплины - изучение основ программирования на языке ассемблера для микропроцессора Intel 8086 (сокращенно - i8086).

Выбор в качестве объекта изучения ассемблера обусловлено следующим. Во-первых, только ассемблер позволяет получить представление о организации и функционировании аппаратуры ЭВМ. Другие языки программирования не предоставляют (или почти не предоставляют) такой возможности. Во-вторых, среди языков ассемблера данный язык является наиболее распространенным. Несмотря на то, что сам процессор i8086 практически стал достоянием компьютерной истории, его машинный язык аппаратно поддерживается в последующих моделях фирмы INTEL. Собственные машинные языки (и языки-ассемблеры) этих моделей включают язык для i8086 в качестве своего подмножества, отличаясь от него большей сложностью, которая делает их мало пригодными для первоначального знакомства с предметом.

Изучение курса “ЭВМ и ПУ” завершается получением допуска к зачету и сдачей экзамена. Для получения зачета требуется успешно выполнить лабораторные и две контрольные работы, подтвердив это соответствующими отчетами. Отчеты по лабораторным работам и по первой контрольной работе предоставляются в виде файлов.

А отчет по второй контрольной работе – на бумажном носителе и в виде файлов. Допускается предоставление отчета по второй контрольной работе полностью в виде файлов.

На носителе создается каталог INFORMAT, имеющий по одному подкаталогу для каждой пересылаемой работы. Имена подкаталогов: LAB1, LAB2, LAB3, CONTR1, CONTR2. Каждая пересылаемая дискета сопровождается письмом, наклейкой или текстовым файлом, содержащим сведения о вас: фамилия, имя, отчество, город, код доступа, пароль.

При выполнении второй контрольной работы, а также при выполнении лабораторных работ используется номер варианта (от 1 до 20). Этот номер рассчитывается по формуле:

$$V = (20 \times K) \text{ div } 100 \quad ,$$

где V – искомый номер варианта (при V = 0 выбирается номер варианта 20);

K – значение двух последних цифр пароля (число от 00 до 99);

div – целочисленное деление (после деления отбрасывается дробная часть).

1) операционная система MS-DOS. Допускается применение ее в самостоятельном режиме или под управлением операционной системы Windows;

2) Norton Commander или другая подобная утилита;
3) DEBUG – отладчик;
4) TASM - транслятор ассемблера i8086;
5) TLINK - редактор связей (компоновщик).

Необходимые сведения о работе с MS-DOS и Commander приведены в приложениях 1 и 2. Применение отладчика, транслятора и редактора связей поясняется в описании лабораторных работ.

2. КРАТКАЯ ПРОГРАММА ЛЕКЦИОННОГО КУРСА

Конспект лекций [1] включает 10 разделов, которые можно разбить на темы:

1) “Начальные сведения” – разделы 1 и 2. Рассматриваются понятия и определения, относящиеся к вычислительным системам, а также излагаются основы представления информации в таких системах;
2) “Аппаратура ЭВМ” – разделы 3, 4 и 5. Рассматриваются структура ЭВМ, организация и работа центрального процессора, логическая организация оперативной памяти. Кроме того, рассматриваются основные типы списков и их применение для организации взаимодействия между подпрограммами;
3) “Операторы ассемблера” – раздел 6. Рассматриваются основные типы операторов обработки данных (арифметические, логические, передачи данных и т.д.), основные типы операторов передачи управления (операторы условных и безусловных переходов, циклов, процедур), адресация и определение данных, а также макрооператоры;
4) “Этапы разработки и преобразования программы” – разделы 7 и 8. Рассматриваются этапы разработки программы: проектирование, кодирование и отладка. А также этапы преобразования программы – трансляция, связывание и загрузка. Особое внимание уделяется модульной организации программы;
5) “Системные программы” – разделы 9 и 10. Рассматривается файловая организация информации во внешней памяти, предоставляемая операционной системой MS-DOS. Излагаются классификация системных программ и их основные особенности.

3. РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

Настоящие методические указания к лабраторным работам конспект лекций содержат информацию, достаточную для выполнения лабораторных и контрольныхработ. Для более подробного рассмотрения излагаемых вопросов рекомендуется обратиться к литературе, список которой приводится ниже.

Книга [1] рекомендуется в качестве дополнительного пособия при выполнении лабораторного курса. Книга [2] рекомендуется для первоначального, а [3] – для углубленного знакомства с ассемблером или в качестве справочного пособия. Для изучения архитектуры i8086 рекомендуется книга [4]. Среди книг, вышедших в последние годы, рекомендуются [5,6,7], хотя можно пользоваться и другими доступными источниками.

1. Нортон П., Соухэ Д. Язык ассемблера для IBM PC. – М., “Компьютер”, 1992, 352 с.
2. Абель П. Язык ассемблера для IBM PC и программирования. – М., “Высшая школа”, 1992, 448 с.
3. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера. – М., “Радио и связь”, 1991, 336 с.
4. Лю Ю., Гибсон Г. Микропроцессоры семейства 8086/8088. – М., ”Радио и связь”, 1987, 512 с.
5. Майко Г.В. Ассемблер для IBM PC. – М., ”Бизнес-Информ”, 1997, 212 с.
6. Юров В., Хорошенко С. Ассемблер: учебный курс. – С.П., “Питер”, 1999, 665 с.
7. Зубков С.В. Ассемблер для DOS, Windows и Unix. – М., ДМК, 1999, 640 с.

4. ЛАБОРАТОРНЫЕ РАБОТЫ

Введение

Данный лабораторный курс состоит из двух частей. В процессе выполнения первой части (работы 1 и 2) производится знакомство с языком, занимающим промежуточное положение между машинным языком и ассемблером. Обладая ассемблерной формой записи кодов операций и регистров, данный язык не имеет ассемблерных псевдооператоров и использует численные (не символьные) адреса. Знакомство с данным языком производится с помощью отладчика Debug и предназначено прежде

всего для изучения механизма выполнения процессором машинных программ. Кроме того, создается основа для последующего написания и отладки ассемблерных программ.

Во второй части курса (работы 3 и 4) производится знакомство с основами программирования на языке ассемблера. При этом объектами рассмотрения являются не только операторы ассемблера, но и методы проектирования и отладки программ.

Лабораторная работа N 1
АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ И ВЫВОД СИМВОЛОВ

Цель работы

Целью настоящей работы является первоначальное знакомство с программой Debug – важнейшим помощником разработчика программ на языке Ассемблер. С помощью этой программы производится анализ и заполнение ячеек регистровой и оперативной памяти, осуществляется пошаговое выполнение программы. Другая цель: знакомство с некоторыми инструкциями Ассемблера, выполняющими арифметические операции, знакомство с инструкциями программного прерывания, а также с инструкциями пересылки данных.

Чтение и заполнение регистров

Почему программа-отладчик называется Debug? "Bugs" (дословно "насекомые") в переводе со слэнга программистов означает "ошибки в программе". Используя Debug для пошагового запуска программы и наблюдая, как программа работает на каждом этапе, мы можем найти ошибки и исправить их. Этот процесс называется отладкой ("debugging"), отсюда и произошло название программы Debug.

З а п у с т и т е Debug, набрав его название после приглашения DOS (которое в этом примере выглядит как "C>"):

C > DEBUG

Debug можно вызвать и с помощью программы Commander, имеющейся на Вашей ЭВМ. Для этого надо найти в каталоге файлов файл DEBUG.COM, установить на него псевдокурсор и нажать клавишу <Enter>.

Дефис “ _ “, который Вы видите в качестве ответа на Вашу команду – это приглашение программы Debug, в то время как "C>" – это приглашение DOS. Это означает, что Debug ждет Вашей команды. Чтобы покинуть Debug и вернуться в DOS, напечатайте "Q" ("Quit") около дефиса и нажмите "Enter". ***П о п р о б у й т е*** выйти и затем обратно вернуться в Debug:

```
_Q
C > DEBUG
```

Мы начнем использование Debug с того, что попросим его показать содержимое регистров центрального процессора (микропроцессора i8086) с помощью команды R (от "Register"):

```
_R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756  SS=3756  CS=3756  IP=0100 NV UP DI PL NZ NA PO
NC  3756:0100  E485  IN  AL,85
```

Возможно, на своем экране вы увидите другие числа во второй и третьей строках. Эти числа зависят от количества памяти. А сейчас обратим внимание на первые четыре регистра, AX, BX, CX и DX, о значениях которых Debug сообщил, что они все равны 0000. Это регистры общего назначения. Остальные регистры SP, BP, SI, DI, DS, ES, SS, CS, IP являются регистрами специального назначения. Так как каждый из 13 регистров i8086 является словом и имеет длину 16 бит, то его содержимое представлено на экране в виде четырехзначного шестнадцатеричного числа.

Команда Debug "R" не только высвечивает регистры. Если указать в команде имя регистра, то Debug поймет, что мы хотим взглянуть на содержимое именно этого регистра и может быть изменить его. Например, мы можем изменить содержимое AX:

```
_R  AX
AX=0000
:3A7
```

Теперь можно убедиться в том, что в регистре AX содержится 3A7h:

```
_R
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA PO NC
3756:0100  E485  IN AL,85
```

Так и есть. Итак, мы можем помещать шестнадцатеричное число в регистр с помощью команды R, указывая имя регистра и вводя его новое значение после двоеточия.

Сложение двух чисел

Теперь мы перейдем к написанию и выполнению с помощью Debug программы на машинном языке. Простейшая такая программа состоит всего из одной машинной инструкции (команды). Допустим, что эта команда выполняет сложение двух чисел, предварительно записанных в регистры.

Допустим, что мы хотим сложить числа 3A7h и 92Ah . Запишем эти числа соответственно в регистры AX и BX, воспользовавшись командой R Debug. Для суммирования содержимого AX и содержимого BX будем использовать машинную инструкцию:

D801

Данная машинная инструкция имеет длину два байта. Старший байт инструкции – D8, а младший – 01. Для лучшего восприятия человеком используется ассемблерная (мнемоническая) форма записи этой же самой инструкции:

ADD AX, BX

Для того чтобы данная машинная инструкция была исполнена центральным процессором (ЦП), необходимо выполнить четыре действия:

1) выбрать место в оперативной памяти (ОП), куда поместить (записать) машинную инструкцию;
2) выполнить запись инструкции на выбранное место;
3) сообщить ЦП о том, где расположена наша машинная инструкция;
4) запустить ЦП для того, чтобы он исполнил инструкцию.

Первое из перечисленных действий заключается в том, что мы должны выбрать из огромного числа ячеек (байтов) ОП всего два соседних байта для размещения нашей двухбайтовой инструкции. Мы не будем сильно "блуждать" по ОП, а ограничимся той областью памяти, которую нам рекомендует использовать Debug. Данная область (сегмент) имеет длину 65536 байт (64 Кбайт). Эта длина обусловлена тем, что 65535 – максимальное число, которое можно записать в шестнадцатибитовое слово.

Где расположен в ОП предоставляемый нам сегмент? Для этого достаточно прочитать содержимое регистра CS, воспользовавшись командой R Debug. Регистр CS называется регистром сегмента кода и содержит номер параграфа, с которого начинается сегмент. Параграф – область ОП длиной 16 байт. Параграф 0 начинается с ячейки 0, параграф 1 – с ячейки l0h, параграф 2 – 20h и т.д. Для того, чтобы получить номер первой ячейки параграфа, достаточно его номер умножить на 16. На практике такое умножение выполняется очень просто – к номеру параграфа в шестнадцатеричном коде (он и содержится в CS) справа приписывается 0.

Из 64 Кбайт сегмента, на который указывает CS, нам надо выбрать всего два байта. В принципе можно взять любое смещение, меньшее чем 65535, относительно начала сегмента. Но чаще всего берут смещение l00h. Ячейки сегмента с меньшими внутрисегментными адресами резервируют для служебной информации, помещаемой туда DOS.

После того, как мы выбрали место для размещения нашей машинной инструкции в ОП, ***з а п и ш и т е*** ее туда с помощью команды Debug E (от "ENTER"), предназначенной для исследования и изменения памяти:

```
_E 100
3756:0100  E4.01
_E 101
3756:0101  85.D8
```

В результате числа 0lh и D8h расположены по адресам 3756:0100 и 3756:0101. Числа E4h и 85h представляют собой старое содержимое

указанных ячеек ОП, которое осталось от ранее выполнявшихся программ. Номер начального параграфа сегмента, который вы увидите, возможно, будет другим, но это различие не будет влиять на нашу программу. Обратите внимание, что младший байт инструкции (0lh) записан нами в ячейку с меньшим адресом, а старший байт (D8h) – с большим адресом. Заметим, что для записи нескольких соседних байтов мы можем использовать всего одну команду E, разделяя пробелом уже записанный байт от следующего:

```
_E 100
3756:0100   E4.01  85.D8
```

Прежде, чем идти дальше, ***п р о в е р ь т е*** результат наших предыдущих действий с помощью команды R:

```
_R
AX=03A7 BX=092A CX=0000 DX=0000 SF=FFEE BP=0000 Sl=0000
DI=0000 DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA
PO NC 3756:0100   01D8   ADD AX,BX
```

Теперь Вам понятно, что содержит последняя строка, выданная Debug. Она содержит: 1) адрес в ОП машинной инструкции: 2) шестнадцатеричный код этой инструкции, причем младший байт инструкции изображен слева, а старший справа (содержимое регистров показывается наоборот – старший байт слева, а младший справа); 3) ее мнемоническое представление. Почему именно нашу инструкцию показал Debug? Ответ заключается в том, что Debug высвечивает ту инструкцию, младший байт которой имеет адрес:

(CS):(IP) ,

где CS - регистр сегмента кода;

IP - указатель инструкции (от "Instruction Pointer");

(r) - содержимое регистра г.

Указатель инструкции IP очень важный специальный регистр, без которого не обходится ни один ЦП. Он содержит внутрисегментное смещение младшего байта той машинной инструкции, которая будет исполняться следующей на ЦП. В процессе выполнения машинной программы ее инструкции сами могут изменять содержимое IP. А пока для этой цели мы будем использовать Debug. После своего запуска Debug всегда записывает в IP 100h. В процессе выполнения нашей машинной программы это значение будет меняться. Пользуясь командой R Debug, мы всегда можем записать в IP требуемое нам значение.

Теперь регистры и ОП готовы для исполнения нашей машинной инструкции. ***П о п р о с и т е*** Debug ее выполнить, используя команду T (от "Trace"), которая выполняет одну инструкцию за шаг, а затем показывает содержимое регистров. После каждого запуска IP будет указывать на следующую инструкцию, в нашем случае будет указывать на 102h. Мы не помещали никакой инструкции в 102h, поэтому в последней строке распечатки мы увидим инструкцию, оставшуюся от предыдущей

программы:

```
_T
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000
DI=0000 DS=3756 ES=3756 SS=3756 CS=3756 IP=0102 NV UP DI PL NZ NA
PO NC 3756:0102   AC   LODSB
```

Вот и все. Регистр AX теперь содержит число CD1h, которое является суммой 3A7h и 92Ah. А регистр IP указывает на адрес 102h, так что в последней строке распечатки регистров мы видим инструкцию, расположенную в памяти по адресу l02h, а не по адресу l00h.

Как отмечалось ранее, указатель инструкции IP вместе с регистром CS всегда указывает на следующую инструкцию, которую нужно выполнить процессору. Если мы опять напечатаем "T", то выполнится следующая инструкция. Но не делайте этого сейчас – ваш процессор может "зависнуть".

Если мы захотим выполнить введенную инструкцию еще раз, то есть сложить 92Ah и CD1h и сохранить новый ответ в AX, то надо объяснить процессору, где найти следующую инструкцию, и чтобы этой следующей инструкцией оказалась та же "ADD AX,BX", расположенная по адресу l00h. Изменить значение регистра IP на l00h можно, используя команду R. После этого:

```
_R
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000
DI=0000 DS=3756 ES=3756 SS=3756 CS=3756 IP=0100 NV UP DI PL NZ NA
PO NC 3756:0100   01D8   ADD   AX,BX
```

П о п р о б у й т е еще раз ввести команду "T" и убедитесь, что регистр AX содержит число 15FBh.

Следовательно, перед тем, как использовать команду T, вам необходимо проверить регистр IP и соответствующую его значению инструкцию, располагаемую в нижней части распечатки (листинга), выдаваемой командой R. В результате вы будете уверены, что процессор выполнит требуемую инструкцию.

Вычитание двух чисел

Мы собираемся написать инструкцию для вычитания BX из AX, так что после двух вычитаний в регистре AX появится результат 3A7h. Тогда мы вернемся к той точке, с которой начали. ***З а п и ш и т е*** с помощью команды E инструкцию вычитания в ОП:

```
_E 100
3756:0100   01.29   D8.D8
```

Листинг регистров (не забывайте установить IP в 100h) должен теперь показать инструкцию "SUB AX,BX", которая вычитает содержимое регистра BX из регистра AX и размещает результат в AX.

В ы п о л н и т е эту инструкцию с помощью команды T. AX должен

содержать CD1. Измените IP так, чтобы он указывал на эту инструкцию, и выполните ее опять (не забывайте сначала проверить инструкции внизу листинга регистров), AX теперь должен содержать 03A7h.

И с п о л ь з у й т е инструкцию SUB, чтобы подтвердить свои знания о представлении отрицательных чисел. Вычтем из 0 (в регистре AX) единицу (в BX). В результате AX должен содержать FFFFh (-1).

Сложение двух байтов

До этого арифметические действия совершались над шестнадцатибитными словами. Но рассматриваемый процессор может выполнять действия и над восьмибитными байтами.

Каждый регистр общего назначения может быть разделен на два байта – старший байт (первые две шестнадцатеричные цифры) и младший байт (следующие две шестнадцатеричные цифры). Название каждого из полученных регистров складывается из первой буквы названия регистра (от "A" до "D"), стоящей перед X в слове, и буквы H для старшего байта или буквы L для младшего. Например, DL и DH – регистры длиной в байт, а DX - длиной в слово.

Проверим байтовую арифметику на инструкции ADD. ***В в е д и т е*** два байта 00h и C4h, начиная с адреса 0l00h. Внизу листинга регистров вы увидите инструкцию "ADD AH,AL", которая суммирует два байта регистра AX и поместит результат в старший байт AH.

Затем ***з а г р у з и т е*** в AX число 0102h. Таким образом, вы поместите 0lh в регистр AH и 02h в регистр AL. Установите регистр IP в l00h, выполните команду T, и вы увидите, что регистр AX теперь содержит 0302. Результат сложения 0lh и 02h будет 03h, и именно это значение находится в AH.

Умножение двух чисел

Инструкция умножения называется "MUL", а машинный код для умножения AX на BX - E3F7h. Так как умножение двух 16-битных чисел может дать 32-разрядный ответ, то инструкция MUL сохраняет результат в двух регистрах - DX и AX. Старшие 16 бит помещаются в регистре DX, а младшие - в AX. Эта комбинация регистров записывается как DX:AX.

В в е д и т е с помощью Debug инструкцию умножения E3F7h по адресу 0l00h и установите AX=7C4Bh и BX=100h. Вы увидите инструкцию в листинге регистров как "MUL BX", без всяких ссылок на регистр AX. При умножении слов процессор i8086 всегда умножает регистр, имя которого вы указываете в инструкции, на регистр AX, и сохраняет ответ в паре регистров DX:AX.

Перед запуском инструкции умножения перемножим l00h и 7C4Bh вручную. Три цифры 100 имеют в шестнадцатеричной системе такой же

эффект, как и в десятичной. Так что умножение на l00h просто добавит два нуля справа от шестнадцатеричного числа. Таким образом, l00h * 7C4B = 7C4B00h. Этот результат слишком длинен для того, чтобы поместиться в одном слове, поэтому мы разбиваем его на два слова 007Ch и 4B00h.

И с п о л ь з у й т е Debug для запуска инструкции. Вы увидите, что DX содержит слово 007Ch, а AX содержит слово 4B00h.

Деление двух чисел

При делении сохраняется как результат, так и остаток от деления.

П о м е с т и т е инструкцию F3F7h по адресу 0100h (и 101h). Как и инструкция MUL, DIV использует пару регистров DX:AX, не сообщая об этом, так что все, что мы видим – это "DIV BX". Загрузите в регистры значения: DX=007Ch и AX=4Bl2h; регистр BX по-прежнему должен содержать 0l00h.

Подсчитаем результат вручную: 7C4B12h/100h=7C4Bh с остатком 12h. После выполнения инструкции деления по адресу 0l00h мы получим в AX результат нашего деления (7C4Bh) и в DX остаток (0012h).

Инструкция программного прерывания

Такая машинная инструкция используется в программе для того, чтобы обратиться за помощью к системному программному обеспечению (в том числе, к DOS). Термин "прерывание" означает, что выполнение нашей программы прерывается (приостанавливается) на время, необходимое для выполнения требуемой системной программы. Инструкция программного прерывания обозначается как INT (от "Interrupt" – прерывание). Инструкция INT для функций DOS имеет вид "INT 21h", в машинном коде 21CDh .

Вывод одного символа

Примером функции DOS, выполнение которой мы можем запросить с помощью инструкции INT 21h, является вывод символа на экран. Для того, чтобы различать функции DOS, которых много, используется регистр AH. При выводе символа в него помещается 02h. В регистр DL заносится код ASCII выводимого символа. В табл. 1 приведены отображаемые (видимые на экране) коды ASCII ("American Standard Code for Information Interchange " – Американский стандартный код для обмена информацией). Допустим, что мы хотим вывести символ A, тогда в регистр DL мы должны поместить число 41h.

Таблица 1. Коды ASCII

Символ ASCII	16-рич код	Символ ASCII	16-рич код	Символ ASCII	16-рич код	Символ ASCII	16-рич код
	20	8	38	P	50	H	68
!	21	9	39	Q	51	I	69
“	22	:	3A	R	52	J	6A
#	23	;	3B	S	53	k	6B
$	24	<	3C	T	54	L	6C
%	25	=	3D	U	55	M	6D
&	26	>	3E	V	56	n	6E
‘	27	?	3F	W	57	o	6F
(	28	@	40	X	58	p	70
)	29	A	41	Y	59	q	71
*	2A	B	42	Z	5A	r	72
+	2B	C	43	[	5B	s	73
,	2C	D	44	\	5C	t	74
-	2D	E	45	]	5D	u	75
.	2E	F	46	^	5E	v	76
/	2F	G	47	_	5F	w	77
0	30	H	48	`	60	x	78
1	31	I	49	a	61	y	79
2	32	J	4A	b	62	z	7A
3	33	K	4B	c	63	{	7B
4	34	L	4C	d	64	|	7C
5	35	M	4D	e	65	}	7D
6	36	N	4E	f	66	~	7E
7	37	O	4F	g	67	DEL	7F

П о д г о т о в ь т е регистры и память для последующего выполнения инструкции INT 21. Для этого в регистры AX и DX запишем с помощью Debug числа 0200h и 0041h, а по адресу 0l00h в ОП запишем 21CDh. После этого можно перейти к выполнению инструкции программного прерывания. Для этого не рекомендуем использовать команду T Debug. Дело в том, что в результате выполнения INT 21 начинает выполняться системная подпрограмма вывода символа, состоящая из многих машинных инструкций. Пошаговое выполнение этой подпрограммы вам скоро наскучит. Но если вы не доведете его до конца, то ваш компьютер

"зависнет". (После того, как вы протрассируете несколько шагов, можно выйти из Debug с помощью команды Q, которая ликвидирует беспорядок.) При выполнении трассировки обратите внимание на то, что изменилось первое число, являющееся составляющей адреса. Это обусловлено тем, что единственная машинная инструкция нашей программы и подпрограмма DOS находятся в разных сегментах ОП.

В этом случае намного удобнее использовать команду Debug G (от "GO"), после которой пишется адрес,на котором мы хотим остановиться :

```
_G 102
A
AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0102 NV UP DI PL NZ NA PO NC
3970:0102  BBE5    MOV  SP,BP
```

DOS напечатал букву A и возвратил затем управление в нашу программу. (Инструкция, размещенная по адресу 102h, осталась от другой программы, поэтому последняя строка вашего листинга может выглядеть по-другому.)

Инструкция завершения программы

Машинная инструкция INT 20h сообщает DOS о том, что мы хотим выйти из нашей программы, и чтобы управление опять вернулось в DOS. В нашем случае эта инструкция вернет управление к Debug, так как мы выполняем нашу программу не непосредственно из DOS, а из Debug.

В в е д и т е инструкцию 20CDh, начиная с адреса l00h, а затем проделайте следующее (не забудьте проверить инструкцию "INT 20h" с помощью команды R):

```
_G 102
Program  terminated  normally
_R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970  ES=3970  SS=3970 CS=3970 IP=0100 NV UP DI PL NZ NA PO NC
3970:0100  CD20   INT 20
_G
Program terminated normally
_R
```

Результат команды G аналогичен результату команды G 102. Любая из этих команд выполняет всю программу (сейчас она состоит всего из одной инструкции – INT 20h) и затем возвращается к началу. Когда мы начали выполнение, IP был установлен в l00h, т.к. мы заново запустили Debug. После выполнения G IP опять содержит l00h.

Мы можем поместить инструкцию INT 20h в конец любой программы для того, чтобы красиво передать управление DOS (или Debug). Для начала поместим ее после инструкции INT 21h и получим программу из двух

инструкций, выполняющую вывод символа на экран. Для этого начиная с адреса l00h ***в в е д и т е*** одну за другой две инструкции – 21CDh и 20CDh.

Когда у нас была только одна инструкция, то мы могли "пролистать" эту инструкцию командой R, но теперь у нас две инструкции. Чтобы увидеть их, воспользуемся командой U (от "Unassemble" – разассемблирование):

```
_U  100
3970:0100  CD21  INT 21
3970:0102  CD20  INT 20
```

Далее идут еще 12 строк листинга, содержащие инструкции, оставшиеся в памяти от предыдущих программ.

П о м е с т и т е в регистр AH значение 02h, а в регистр DL код любого символа, например код символа F – 46h. Затем введите команду G, чтобы увидеть символ на экране:

```
_G
F
Program terminated normally
```

До этого мы вводили инструкции программы в виде чисел, например 21CDh. Но это слишком тяжелая работа и избавиться от нее помогает команда Debug A (от "Assemble" – ассемблирование). Эта команда помогает вводить мнемонические (человекочитаемые) инструкции. Применим команду A для ввода нашей программы:

```
_A  100
3970:0100   INT  21
3970:0102   INT  20
3970:0104
```

Команда A сообщает Debug о том, что мы хотим ввести инструкции в мнемонической форме, а число 100 в команде означает, что ввод инструкций начинается с ячейки l00h.

Пересылка данных между регистрами

До сих пор мы записывали требуемые числа в регистры с помощью команды R Debug. Но обычно это делает инструкция самой программы – MOV. Эта же инструкция выполняет пересылку чисел между регистрами.

П о м е с т и т е 1234h в AX (12h в регистр AH и 34h в AL) и ABCDh в DX (ABh в DH и CDh в DL). С помощью команды A введите инструкцию MOV AH,DL. Эта инструкция пересылает (копирует) число из DL в AH, AL при этом не используется. Если вы протрассируете эту строку, то увидите, что AX=CD34h и DX=ABCDh. Изменился только AH. Теперь он содержит копию числа из DL.

Инструкция MOV пересылает число из второго регистра в первый, и по этой причине мы пишем AH перед DL. Машинный код данной инструкции D488h. Существуют другие формы этой же инструкции MOV. Они имеют другие машинные коды и выполняют другие операции пересылки.

Например, следующая инструкция (C389) выполняет пересылку не байтов, а слов между двумя регистрами AX и BX:

```
3970:0100   89C3    MOV  BX, AX
```

Следующая форма инструкции MOV записывает значение числа в регистр, не используя другой регистр-источник:

```
3970:0100    B402   MOV  AH, 02
```

Эта инструкция загружает число 02h в регистр AH. Старший байт инструкции, 02h является числом, которое мы хотим загрузить**.** ***З а п и ш и т е*** эту инструкцию в ОП и выполните ее. Затем загрузите в AH другое число: с помощью команды "E 101" измените старший байт, чтобы он был равен, например, C1h .

Сложим все части вместе и построим длинную программу. Она будет печатать звездочку *, выполняя все операции сама, не требуя от нас установки регистров (AH и DL). Программа использует инструкции MOV для того, чтобы установить регистры AH и DL перед выполнением инструкции INT 21h, выполняющей вызов функции DOS:

```
15AC:0100 B402   MOV  AH, 02
15AC:0102 B22A   MOV  DL, 2A
15AC:0104 CD21   INT    21
15AC:0106 CD20   INT    20
```

В в е д и т е программу и проверьте ее командой U 100. Убедитесь, что IP указывает на ячейку l00h. Запустите программу командой G. В итоге на экране должен появиться символ *.

Теперь у нас есть законченная программа. Запишем ее на диск в виде .com-файла для того, чтобы мы могли запускать ее прямо из DOS, просто набрав ее имя. Так как у программы пока нет имени, то мы должны его присвоить.

Команда Debug N (от "Name") присваивает файлу имя перед записью на диск. ***Н а п е ч а т а й т е*** **:**

```
_N   Writestr.com
```

Эта команда не запишет файл на диск – она только назовет его Writestr.com .

Далее мы должны сообщить Debug о том, сколько байт занимает программа, для того, чтобы он знал размер файла. Если вы посмотрите на разассемблированный листинг программы, то увидите, что каждая инструкция в нем занимает два байта (в общем случае это не выполняется). У нас четыре инструкции, следовательно программа имеет длину восемь байт.

Полученное число байт надо куда-то записать. Для этого Debug использует пару регистров BX:CX, и поэтому, поместив 8h в CX, мы сообщим Debug о том, что программа имеет длину в восемь байт. BX должен быть предварительно установлен в ноль.

После того, как мы установили имя и длину программы, мы можем

записать ее на диск с помощью команды Debug W (от "Write"):

```
_W
Writing 0008 bytes
```

Теперь на диске есть программа Writestr.com, а мы с помощью Q покинем Debug и посмотрим на нее. ***И с п о л ь з у й т е*** команду DOS Dir , чтобы увидеть справочную информацию о файле:

```
C >DIR   Writestr.com
Volume in drive C has no label
Directory  of  C:\
WRITESTR.COM   8   6-30-93  10:05a
1 File (S)     18432  bytes  free
```

Листинг директории сообщает, что Writestr.com находится на диске С и его длина составляет восемь байт. Чтобы загрузить программу, наберите writestr в ответ на приглашение DOS и нажмите "Enter". Вы увидите *.

Если мы хотим запустить свою .com-программу не из DOS, а из DEBUG, то запуск DEBUG следует выполнить вместе с требуемым загрузочным модулем. Пример такого запуска: DEBUG WRITESTR.COM. После этого с данной программой можно работать так, как будто мы создали ее только что с помощью DEBUG, а не считали с диска. Для сохранения скорректированной программы на диске следует выполнить те же операции, что и для нового файла.

Вывод на экран строки символов

Функция номер 02h для прерывания INT 21h печатает один символ на экране. Другая функция, номер 09h, печатает целую строку и прекращает печатать, когда находит символ "$".

Поместим строку в память, начиная с ячейки 200h, чтобы строка не перепуталась с кодом самой программы. ***В в е д и т е*** следующие числа, используя команду E 200:

```
48  65  6C  6C      48  65  6C  6C  6F  2C 20  44
6F  2C  20  44      4F  53  20  68  65  72  65  2E
4F  53  20  68      24
65  72  65  2E
24
```

Последнее число 24h является ASCII–кодом для знака $, и оно сообщает DOS, что это конец строки символов. Теперь посмотрим, что сообщает эта строка, запустив следующую программу:

```
15AC:0100  B409         MOV   AH, 09
15AC:0102  BA0002       MOV   DX, 0200
15AC:0105  CD21         INT    21
15AC:0107  CD20         INT    20
```

200h – адрес строки, которую мы ввели, а загрузка 200h в регистр DX

сообщает DOS о том, где ее искать. ***П р о в е р ь т е*** программу командой U и затем запустите ее командой G:

```
_G
Hello, DOS here.
Program terminated normally
```

Команда Debug D (от "Dump") дампирует (выводит содержимое) памяти на экран. Это похоже на действия, совершаемые командой U при распечатке инструкций. Подобно U поместите после D адрес, чтобы сообщить Debug, откуда начинать дамп.

Н а б е р и т е команду "D 200". Она выводит содержимое участка памяти, в котором хранится только что введенная строка:

```
_D  200
15AC:0200 48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E Hello, DOS here.
15AC:0210  24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8 $J.U..0……
```

После каждого числа, обозначающего адрес (как 15AC:0200 в примере), мы видим 16 пар шестнадцатеричных чисел, вслед за которыми записаны 16 ASCII–символов для этих пар (байтов). Например, в первой строке записаны символы, которые вы ввели. Символ $ является первым символом в следующей строке, остальная часть строки представляет собой беспорядочный набор символов.

Точка "." в окне ASCII означает, что это может быть как точка, так и специальный символ, например, греческая буква " pi ". Команда Debug D выдает только 96 из 256 символов символьного набора IBM PC, поэтому точка используется для обозначения остальных 160 символов. Часть специальных символов представляет собой прописные и строчные буквы русского алфавита. Соответствующие шестнадцатеричные коды приведены в табл. 2.

Теперь запишем программу, выводящую строку на экран, на диск. Программа начинается со строки l00h, и из выполненного дампа памяти можно видеть, что символ, следующий за знаком $, заканчивающим нашу строку, расположен по адресу 211h. ***С о х р а н и т е*** разность 211h-100h в регистре CX, опять установив BX в ноль. Используйте команду N, чтобы дать имя программе (добавьте расширение .com, чтобы запускать программу прямо из DOS), и затем командой W запишите программу и данные в дисковый файл.

Таблица 2. **Коды букв русского алфавита**

Символ	Код(16)	Символ	Код(16)	Символ	Код(16)	Символ	Код(16)
А	80	Р	90	а	А0	р	Е0
Б	81	С	91	б	А1	с	Е1
В	82	Т	92	в	А2	т	Е2
Г	83	У	93	г	А3	у	Е3
Д	84	Ф	94	д	А4	ф	Е4
Е	85	Х	95	е	А5	х	Е5
Ж	86	Ц	96	ж	А6	ц	Е6
З	87	Ч	97	з	А7	ч	Е7
И	88	Ш	98	и	А8	ш	Е8
Й	89	Щ	99	й	А9	щ	Е9
К	8А	Ъ	9А	к	АА	ъ	ЕА
Л	8В	Ы	9В	л	АВ	ы	ЕВ
М	8С	Ь	9С	м	АС	ь	ЕС
Н	8D	Э	9D	н	AD	э	ED
О	8Е	Ю	9Е	о	АЕ	ю	ЕЕ
П	8F	Я	9F	п	AF	я	EF

Задание

Р а з р а б о т а й т е с помощью Debug программу, выполняющую вывод на экран текстового сообщения и последующее вычисление выражения:

Y=[(X1+X2)X3-X4]/X5 ,

где X1 - X5 - десятичные целые числа, взятые в соответствии с номером вашего варианта из таблицы 3.

Структура выходного сообщения программы:

“Программа вычисления выражения Y=[(X1+X2)X3-X4]/X5 , где X1=..., X2=..., X3=..., X4=..., X5=...”

Вместо точек должны выводиться заданные числа (в шестнадцатеричной системе).

Результат вычисления выражения программа помещает в регистры AX и DX. Поэтому этот результат можно наблюдать только при запуске программы из DEBUG (при возврате в DOS содержимое регистров теряется).

Таблица 3

К	X1	X2	X3	X4	X5	К	X1	X2	X3	X4	X5
1	235	314	11	2320	13	11	417	125	14	3181	11
2	513	248	17	4453	12	12	317	373	13	1920	12
3	197	372	12	3838	17	13	550	241	11	2720	15
4	254	418	14	4118	21	14	632	193	12	3593	21
5	349	517	11	5314	19	15	249	431	17	4111	13
6	267	149	15	2773	14	16	391	463	14	5320	22
7	435	317	13	3815	15	17	561	323	13	6213	23
8	561	273	16	2584	20	18	244	395	15	4713	14
9	321	491	12	4511	13	19	139	456	19	5334	17
10	634	124	19	8416	24	20	286	293	16	4811	21

Результат выполнения работы оформляется в виде .com-файла, помещаемого для пересылки в каталог LAB1.

Примечание 1. Загрузка в регистры заданных чисел (преобразованных вручную в шестнадцатеричную систему) должна производиться только с помощью инструкций MOV.

Примечание 2. Рекомендуется выполнить проверку результата выполнения программы путем сравнения его с результатом ручного счета. Так как при ручном счете используется десятичная система счисления, то перед сравнением результатов их необходимо записать в одной и той же системе.

Лабораторная работа N 2
ВВОД-ВЫВОД ЧИСЕЛ

Цель работы

В процессе выполнения работы решается практически важная задача вывода чисел на экран и их ввода с клавиатуры. Данная задача решается в следующей последовательности. Во-первых, рассматривается вывод на экран двоичного числа в виде последовательности единиц и нулей. Во-вторых, решается задача вывода на экран шестнадцатеричных чисел. В-

третьих, рассматривается ввод шестнадцатеричных чисел с клавиатуры.

В ходе работы производится знакомство с очень важными понятиями флагов состояния, стека и процедуры. Изучаются инструкции для работы с этими объектами, а также инструкции сдвига, цикла, условных переходов и некоторые другие.

Одной из целей работы является развитие навыков алгоритмизации задач и отладки программ.

Флаг переноса

Если выполнить сложение чисел 1 и FFFFh, то получим 10000h. Это число не может быть записано в шестнадцатибитное слово, т.к. в нем помещаются только четыре шестнадцатеричные цифры. Единица в результате называется переполнением. Она записывается в специальную ячейку, называемую флагом переноса CF (от "Carry Flag"). Флаг содержит число, состоящее из одного бита, т.е. содержит или единицу или ноль. Если флаг содержит единицу, то говорят, что он "установлен", а если ноль – "сброшен".

В ы п о л н и т е загрузку чисел 1 и FFFFh в регистры BX и AX и запишите в память инструкцию ADD AX,BX. После этого протрассируйте инструкцию ADD. В конце второй строки распечатки, полученной с помощью команды R Debug, вы увидите восемь пар букв. Последняя пара выглядит как CY (от "Carry Ye" - перенос есть), т.е. флаг переноса установлен.

У с т а н о в и т е IP в 100h и прибавьте единицу к нулю в AX, повторив трассировку инструкции сложения. Флаг переноса переустанавливается в каждой операции сложения, и так как на этот раз переполнения не будет, то флаг будет сброшен. С помощью команды R проверьте, что в качестве состояния флага CF листинг содержит NC ("от No Carry" – нет переноса).

Циклический сдвиг

Допустим, что нам надо выполнить вывод на экран двоичного числа. За шаг мы печатаем только один символ, и нам надо произвести выборку всех битов двоичного числа, одного за другим, слева направо. Например, пусть требуемое число есть 10000000b. Если мы сдвинем весь этот байт влево на одну позицию, помещая единицу во флаг переноса и добавляя ноль справа, и затем повторим этот процесс для каждой последующей цифры, во флаге переноса будут по очереди содержаться все цифры нашего двоичного числа.

Инструкция RCL (от "Rotate Carry Left" – циклический сдвиг влево с переносом) сдвигает крайний левый бит во флаг переноса (в примере это 1), в то время как бит, находившийся до этого во флаге переноса, сдвигается в

крайне правую позицию (т.е. в нулевой бит). В процессе сдвига все остальные биты сдвигаются влево. После определенного количества циклических сдвигов (17 для слова, 9 для байта) биты возвращаются на их начальные позиции, и вы получаете исходное число.

В ы п о л н и т е с помощью Debug размещение по адресу 100h инструкции RCL BL,1 , которая циклически сдвигает байт в BL влево на один бит, используя флаг переноса. Поместите в регистр BX число B7h и протрассируйте эту инструкцию несколько раз. Убедитесь, что после 9 циклов регистр BX содержит опять B7h.

Как напечатать двоичное значение флага переноса? Из таблицы кодов ASCII видно, что символ “0” есть 30h, а символ "1" есть 31h.. Таким образом, сложение флага переноса и 30h дает символ "0", когда флаг сброшен и символ “1”, когда он установлен. Для выполнения такого сложения удобно использовать инструкцию ADC (от “Add with Carry” - сложение с переносом). Эта инструкция складывает три числа: два числа, как и инструкция ADD, ПЛЮС один бит из флага переноса.

П о м е с т и т е в память после инструкции RCL BL,1 инструкцию ADC DL,30 , которая выполнит сложение содержимого DL (0), 30h и флага переноса, поместив результат в DL. Записав далее инструкции, обеспечивающие вывод символа на экран и завершение программы, получим программу, выполняющую печать старшего бита регистра BL:

```
MOV     DL, 00
RCL     BL, 1
ADC     DL, 30
MOV     AH, 02
INT     21
INT     20
```

В ы п о л н и т е эту программу для обоих значений старшего бита BL. Для записи в BX используйте команду R Debug.

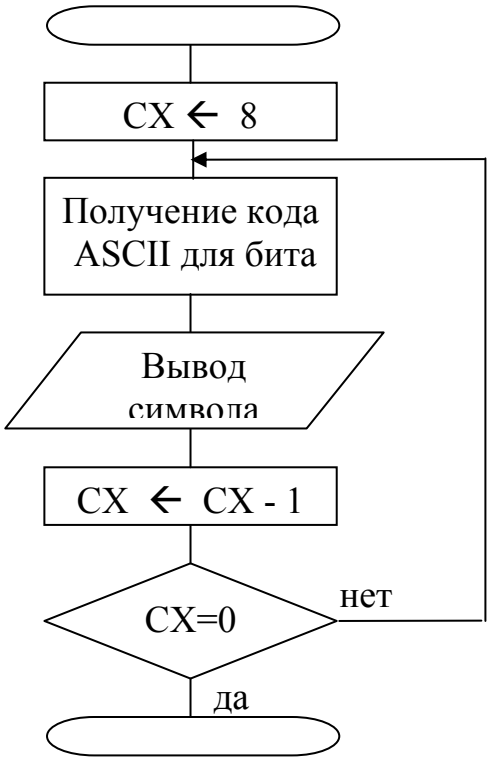
Организация циклов

Если мы хотим распечатать все биты BL, то мы должны повторить операции циклического сдвига и распечатки флага переноса CF восемь раз (число битов в BL). Неоднократное повторение одних и тех же операций называется циклом.

Соответствующий алгоритм приведен на рис. 1. На первом этапе переменной CX присваивается значение 8. Именно столько раз мы хотим повторить выполнение этапов, образующих “тело цикла”. Переменная CX называется “счетчиком повторений”. Тело цикла образуют этапы “Получение кода ASCII для бита” и “Вывод символа”. После того, как тело цикла выполнено, значение CX уменьшается на 1. На следующем этапе

алгоритма значение CX сравнивается с 0. Если это значение ненулевое, то делается возврат для повторения тела цикла. Иначе – выполняется этап алгоритма, расположенный после цикла. На рис. 1 это этап “Завершение алгоритма”.

Для организации циклов используются специальные машинные инструкции. Одной из них является инструкция LOOP. Она записывается в конце цикла, т.е. после тех инструкций, выполнение которых следует повторить. Данная инструкция имеет один операнд – адрес первой из повторяемых инструкций. Счетчик повторений цикла содержится в регистре CX. Данный регистр используется потому, что буква C в названии регистра CX означает “счетчик” (от “Count”). CX может использоваться также как регистр общего назначения.



CX - счетчик повторений

Рис. 1. Алгоритм вывода на экран содержимого байта

Выполнение инструкции LOOP сводится к следующему. Во-первых, она вычитает из текущего содержимого регистра CX единицу. Во-вторых, если полученное значение CX не равно 0, делается переход по адресу, заданному в качестве операнда инструкции LOOP. Таким образом, наличие данной инструкции обеспечивает реализацию двух этапов алгоритма, приведенного на рис. 1.

В качестве примера применения инструкции LOOP приведем программу, выполняющую вывод на экран четырех звездочек:

```
100     MOV     AH, 02
102     MOV     DL, 2A
104     MOV     CX, 4
107     INT     21
109     LOOP    107
10B     INT     20
```

В ы п о л н и т е трассировку данной программы, наблюдая за содержимым регистров IP и CX . При этом следует помнить, что не следует использовать команду T для инструкции INT. При достижении этой инструкции следует набрать команду G d , где d - адрес в памяти инструкции, следующей за инструкцией INT. Эта команда сообщит Debug о том, что надо продолжить выполнение программы до того момента, когда IP достигнет значения введенного адреса. Таким образом Debug будет выполнять инструкцию INT без трассировки и останавливаться при достижении инструкции, следующей за INT. Адрес d называется точкой останова. При достижении инструкции INT 20 вводится команда G.

При применении команды G d необходимо знать адрес d. Исключить трассировку при достижении инструкции INT проще, если использовать команду Debug P (от "Proceed" – переходить, продолжать). Эта команда является удобным средством обхода инструкций, вызывающих подпрограммы DOS.

В ы п о л н и т е написание и ввод в память программы вывода двоичного содержимого байта (содержится в регистре BL) на экран (алгоритм на рис. 1).

Отладка программы

Она включает поиск ошибок в программе (тестирование программы) и их исправление. Пока наши программы достаточно просты, и каждая из них включает всего одну подпрограмму. Что касается отладки программ, состоящих из нескольких подпрограмм, то она будет рассматриваться позднее. Пока лишь заметим, что такая отладка может рассматриваться как последовательность отладок подпрограмм.

Тестирование программы выполняется при различных значениях ее входных данных. Если очередной прогон программы показал наличие в ней ошибки, то производится ее поиск. Как раз для такого поиска и предназначена трассировка программы.

Если программа длинная, то ее пошаговая трассировка не очень удобна. В этом случае сначала желательно локализовать ошибку, определив содержащий ее фрагмент программы. Далее этот фрагмент исследуется

более подробно. Для локализации ошибки мы выбираем несколько точек останова. Для выбора этих адресов, разбивающих программу на фрагменты, используется листинг программы, а также ее блок-схема. Далее, с помощью команды G d производится анализ работы программы в каждой из точек останова. Если в очередной точке останова результаты работы программы неверны, то данная точка завершает искомый фрагмент.

В ы п о л н и т е отладку введенной ранее в память программы вывода двоичного содержимого байта. Тестирование программы проведите с помощью команды G, предварительно загружая с помощью команды R в регистр BX различные пары шестнадцатеричных цифр. (Debug не имеет команд для работы с однобайтовыми регистрами и поэтому BL заполняется как часть BX.) В случае обнаружения ошибки выполните пошаговую трассировку, или используйте точки останова.

Флаги состояния

Кроме флага переноса CF существуют другие флаги состояния. Рассмотрим три из них, которые описывают результат последней арифметической операции.

Допустим, что мы выполнили инструкцию вычитания SUB. Одним из флагов состояния, устанавливаемых в зависимости от результата этой инструкции, является флаг нуля ("Zero Flag"). Если результат инструкции SUB есть 0, то флаг нуля будет установлен в 1. На распечатке регистров это значение обозначается как ZR (от "Zero" – нуль). Если результат арифметической операции не равен нулю, то флаг нуля сбрасывается в 0 – NZ (от "Not Zero" – не ноль).

В в е д и т е в память инструкцию SUB AX,BX. Протрассируйте ее с одинаковыми и с разными числами в регистрах AX и BX, наблюдая за состоянием флага нуля (ZR или NZ).

Флаг знака ("Sign Flag") принимает значение 1 (на распечатке регистров NG – от "Negative"), если результат предыдущей арифметической операции отрицательный. Если результат неотрицательный (ноль или положительный), то флаг знака принимает значение 0 (на распечатке PL – "Plus"). ***В ы п о л н и т е*** трассировку инструкции SUB AX,BX при разных содержимых AX и BX и наблюдая за флагом знака.

Флаг переполнения устанавливается в 1 в том случае, если знаковый бит изменился в той ситуации, когда этого не должно было произойти. Например, если мы сложим два положительных числа 7000h и 6000h, то получим отрицательное число D000h (представление в дополнительном коде числа –12288). Это ошибка, т.к. результат переполняет слово. На листинге регистров переполнение обозначается как OV ("Overflow" – переполнение). Если предыдущая арифметическая инструкция не дала переполнения, то флаг сбрасывается в 0. На распечатке регистров это значение обозначается как NV ("No Overflow"). ***П р о в е р ь т е***

установку флага переполнения протрассировав инструкцию SUB или ADD.

Использование инструкции SUB для сравнения двух чисел неудобно, т.к. эта инструкция производит изменение первого из чисел. Другая инструкция, CMP (от “Compare” – сравнение), производит сравнение двух чисел без их изменения. Результат сравнения используется только для установки флагов.

З а г р у з и т е в регистры AX и BX одинаковые числа, например F5h и протрассируйте инструкцию CMP AX,BX. При этом убедитесь, что установлен флаг нуля (ZR), но оба регистра сохранили свое значение – F5h.

Инструкции условного перехода

Флаги состояния устанавливаются для того, чтобы можно было менять ход выполнения программы в зависимости от текущей ситуации. Анализ флагов состояния и соответствующие переходы в программе выполняют инструкции, называемые инструкциями условного перехода.

Инструкция JZ (от "Jump if Zero" – перейти, если ноль) проверяет флаг нуля, и если он установлен (ZR), то выполняется переход на новый адрес. Таким образом, если мы вслед за инструкцией SUB напишем, например, JZ 15A, то нулевой результат вычитания означает, что ЦП начнет выполнять инструкцию, находящуюся по адресу 15A, а не следующую по порядку инструкцию.

Противоположной по отношению к JZ является инструкция JNZ ("Jump if Not Zero" – перейти, если не ноль). В следующей простой программе из числа вычитается единица до тех пор, пока в результате не получится ноль:

```
100     SUB     AL,01
102     JNZ     100
104     INT     20
```

П о м е с т и т е небольшое число в AL и протрассируйте программу, чтобы увидеть, как работает условное ветвление. При достижении последней инструкции введите команду G.

Инструкция условного перехода JA (от "Jump if Above" – перейти, если больше) осуществляет переход на указанный в инструкции адрес, если по результатам предыдущей инструкции флаг переноса CF сброшен (на листинге CF = NC). Флаг нуля ZF также должен быть сброшен. Данная инструкция обычно записывается сразу за инструкцией сравнения (CMP) двух беззнаковых чисел. Если первое сравниваемое число больше второго, то инструкция JA осуществляет переход.

Вывод на экран одной шестнадцатеричной цифры

Любое число между 0 и Fh соответствует одной шестнадцатеричной цифре. Переведя выбранное число в ASCII-символ, можно его напечатать. ASCII-символы от 0 до 9 имеют значения от 30h до 39h; символы от A до F, однако, имеют значения от 41h до 46h. В результате переход в ASCII будет затруднен из-за наличия двух групп чисел (от 0 до 9 и от Ah до Fh), так что мы должны обрабатывать отдельно каждую группу. На рис.2 приведена блок-схема программы, выполняющей вывод на экран шестнадцатеричной цифры.

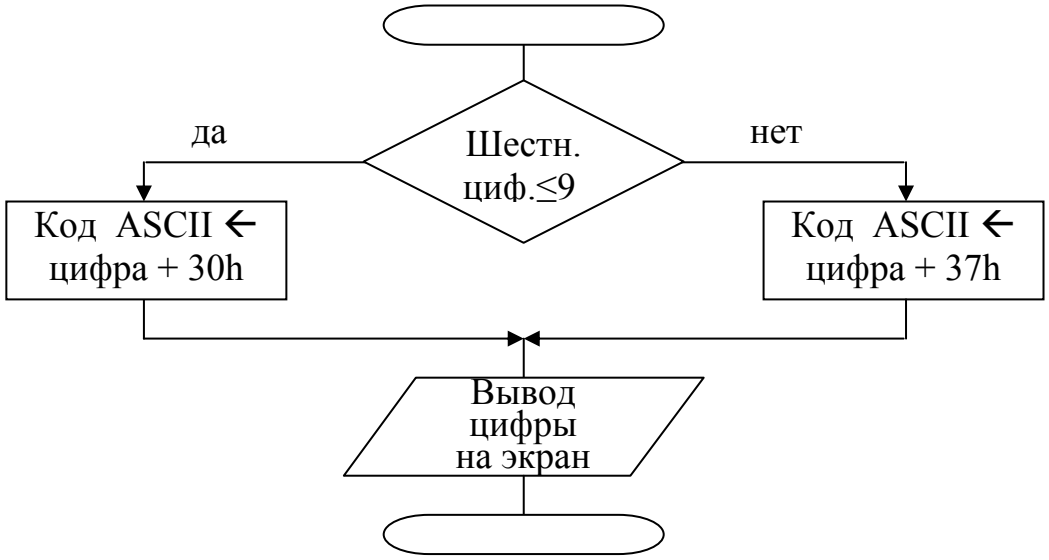


Рис. 2**.** Алгоритм программы вывода одной шестнадцатеричной цифры

Текст программы вывода шестнадцатеричной цифры:

```
100     MOV     DL, BL
102     CMP     DL, 09
105     JA      10C
107     ADD     DL, 30
10A     JMP     10F
10C     ADD     DL, 37
10F     MOV     AH, 02
111     INT     21
113     INT     20
```

Для передачи значения шестнадцатеричной цифры на вход программы используется регистр BL. Инструкция CMP вычитает два числа ((DL)–9h), чтобы установить флаги, но она не изменяет регистр DL. Поэтому, если содержимое DL больше чем 9, инструкция JA 10C осуществляет переход к

инструкции по адресу 10C.

З а п и ш и т е приведенную выше программу в ОП и протрассируйте ее**,** предварительно записав в BL шестнадцатеричное число, состоящее из одной цифры. Не забывайте использовать или команду G с указанием точки останова, или команду P, когда запускаете инструкции INT. Затем проверьте правильность работы программы, используя команду G, предварительно загружая в BX граничные данные: 0; 9; Ah и Fh .

Общее правило: при тестировании любой программы рекомендуется проверить граничные данные.

Вывод старшей цифры двузначного шестнадцатеричного числа

Одна шестнадцатеричная цифра занимает четыре бита (четыре бита часто называют полубайтом или тетрадой). Двузначное шестнадцатеричное число занимает восемь бит (один байт). Это может быть, например, регистр BL. При выводе числа на экран сначала выводится старшая цифра, а затем – младшая. Соответствующий укрупненный алгоритм приведен на рис. 3. При этом детальный алгоритм каждого из двух этапов “Вывод цифры на экран” приведен на рис. 2.

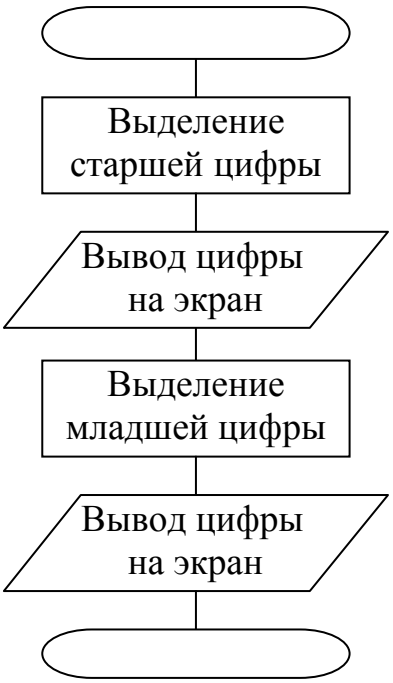


Рис. 3**.** Алгоритм вывода двузначного шестнадцатеричного числа

Рассмотрим этап "Выделение старшей цифры". В результате него содержимое старшего полубайта должно быть переписано (сдвинуто) в младший полубайт. Несмотря на то, что нам надо выполнить сдвиг вправо,

вспомним инструкцию RCL, которая циклически сдвигает байт или слово влево через флаг переноса. Ранее мы использовали инструкцию RCL BL,1 , в которой единица есть сообщение для ЦП о том, что надо сдвинуть BL на один бит. Мы можем осуществить циклический сдвиг более чем на один бит, но мы не можем написать инструкцию RCL BL,2 . Для циклического сдвига необходимо поместить счетчик сдвигов в регистр CL, который используется здесь также, как регистр CX применялся инструкцией LOOP при определении числа повторений цикла. Так как не имеет смысла осуществлять циклический сдвиг более чем 16 раз, то для записи числа сдвигов вполне подойдет восьмибитовый регистр CL.

Для сдвига старшего полубайта вправо на четыре бита будем использовать инструкцию сдвига SHR (“Shift Right" – логический сдвиг вправо). Данная инструкция не только выполняет сдвиг вправо, но и записывает в освобождающиеся старшие биты нули. В этом проявляется разница между терминами "логический" и "циклический", так как инструкция циклического сдвига записывает в освобождающиеся биты содержимое флага переноса. Что касается выталкиваемых младших битов байта (или слова), то они по очереди записываются во флаг переноса аналогично циклическому сдвигу.

З а г р у з и т е числа 4 в CL и 5Dh в DL, а затем введите и протрассируйте следующую инструкцию сдвига:

```
100     SHR     DL, CL
```

DL должен теперь быть равным 05h, т.е. содержит в младшем полубайте старшую цифру числа 5Dh.

Реализацию этапа "Выделение старшей цифры" осуществляют инструкции:

```
MOV     DL, BL
MOV     CL, 04
SHR     DL, CL
```

П о м е с т и т е эти инструкции в ОП, дополнив их инструкциями этапа "Вывод цифры на экран". При этом не забудьте скорректировать адреса переходов. (Можно записать программу со старыми адресами, а затем скорректировать инструкции переходов). Выполните программу, предварительно загрузив в регистр BL любую пару шестнадцатеричных цифр.

Вывод младшей цифры двузначного шестнадцатеричного числа

Для вывода младшей цифры достаточно обнулить старший полубайт в восьмибитовом регистре, содержащем пару шестнадцатеричных цифр.

Обнуление любых битов в байте или в слове удобно выполнить, используя инструкцию AND (логическое "И"). Данная инструкция побитно сравнивает два заданных в ней байта (слова). Если в обоих байтах соответствующий бит имеет значение "1", то и в результирующий байт на

место соответствующего бита записывается 1. Если хотя бы один из сравниваемых битов имеет значение "0", то и результирующий бит принимает нулевое значение. Результирующий байт записывается на место первого из сравниваемых байтов. Например, инструкция AND BL,CL последовательно выполняет операцию AND сначала над битами 0 регистров BL и CL, затем над битами 1, битами 2 и т.д., и помещает результат в BL.

Выполняя операцию AND над 0Fh и каким-либо байтом, мы можем обнулить старший полубайт этого байта:

```
        1011 0101
AND     0000 1111
        0000 0101
```

Следующие две инструкции реализуют этап "Выделение младшей цифры":

```
MOV   DL, BL
AND   DL, 0F
```

З а п и ш и т е в память программу для вывода младшей цифры. Протестируйте эту программу, загружая в BL различные пары шестнадцатеричных цифр. Далее запишите в память всю программу вывода на экран двузначного шестнадцатеричного числа и протестируйте ее. (Не забудьте при этом скорректировать адреса переходов во второй части программы, а также исключить первую инструкцию INT 20).

Ввод одной шестнадцатеричной цифры

Для того, чтобы ввести с клавиатуры в программу ASCII-символ, можно воспользоваться инструкцией программного прерывания INT 21h с функцией номер 1. Вызванная в результате данной инструкции подпрограмма DOS помещает ASCII-символ, соответствующий нажатой клавише, в регистр AL.

В в е д и т е инструкцию INT 21h по адресу l00h, поместите номер функции 1 в регистр AH, а затем запустите инструкцию с помощью G 102 либо P. В результате DOS переходит в состояние ожидания нажатия вами клавиши (на экране вы видите мерцающий курсор). Нажмите любую клавишу, соответствующую шестнадцатеричной цифре (0– F). Убедитесь, что в результате регистр AL содержит соответствующий код ASCII.

При преобразовании ASCII-символа, который содержится в регистре AL, в шестнадцатеричную цифру, решается задача, обратная той, которую мы решали при выводе цифры на экран. На рис. 4 приведена блок-схема программы, которая выполняет ввод цифры с клавиатуры в регистр AL.

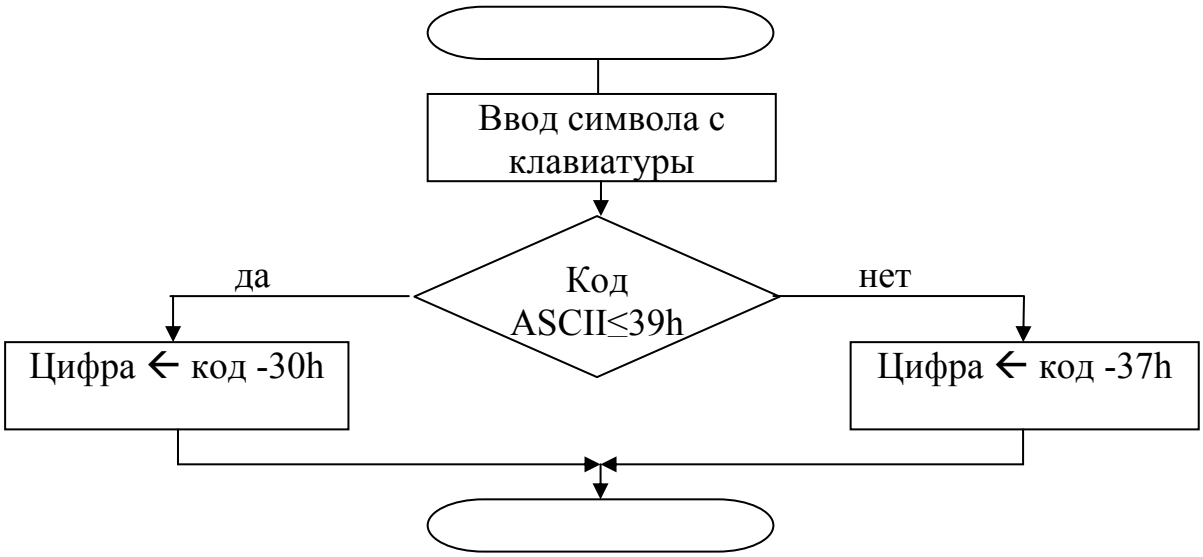


Рис. 4. Алгоритм ввода шестнадцатеричной цифры

З а п и ш и т е текст программы ввода шестнадцатеричной цифры и поместите его в память. Для программирования условия можно использовать не только уже знакомую нам инструкцию условного перехода JA (перейти, если больше), но и обратную ей инструкцию JBE (перейти, если меньше или равно). Обе инструкции используются после сравнения беззнаковых величин, каковыми коды ASCII и являются.

Так как результат данной программы содержится в регистре AL, то этот регистр необходимо проанализировать прежде, чем исполнится инструкция INT 20 (Debug восстанавливает регистры после этой инструкции). Поэтому для запуска программы используйте команду G d, где d – смещение инструкции INT 20.

Выполните программу несколько раз, нажимая не только клавиши, соответствующие шестнадцатеричным цифрам, но и другие клавиши. При этом убедитесь, что программа реагирует на неправильное нажатие клавиши точно так-же, как и на правильное. В дальнейшем этот недостаток будет устранен.

Ввод двузначного шестнадцатеричного числа

Такой ввод можно осуществить следующим образом. Введем старшую цифру числа, записав ее в четыре младших бита регистра DL. Далее умножим регистр DL на 16, в результате чего цифра переместится в

старшие четыре бита DL. После этого введем с клавиатуры младшую цифру числа и просуммировав AL с DL, получим в DL все двузначное шестнадцатеричное число. Соответствующий алгоритм приведен на рис. 5.

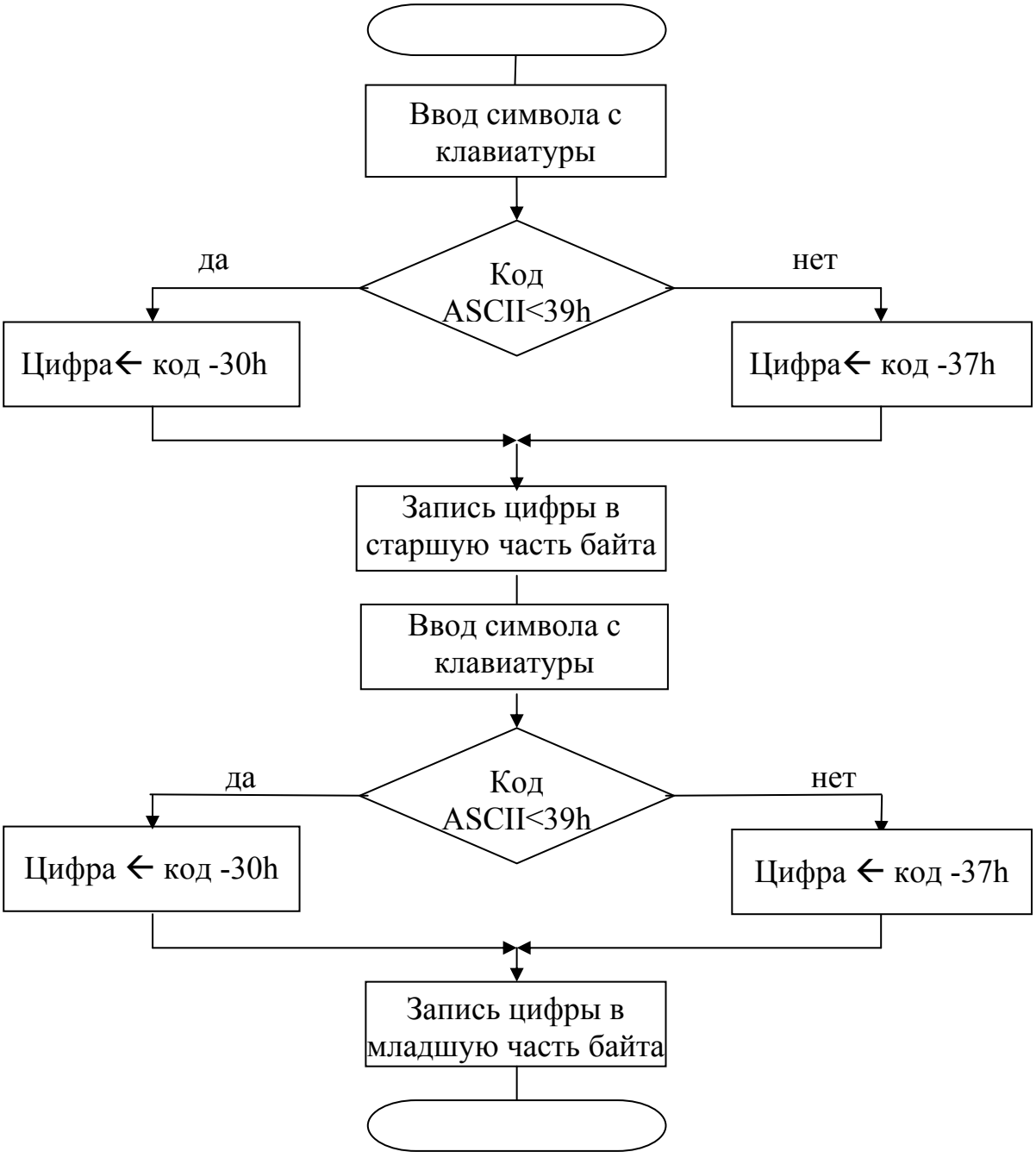


Рис. 5. Алгоритм ввода двузначного шестнадцатеричного числа

З а п и ш и т е программу ввода с клавиатуры (в регистр DL) двузначного шестнадцатеричного числа. (Для сдвига регистра DL влево используйте рассмотренную ранее инструкцию SHL.) Введите данную программу в память и протрассируйте ее при различных парах чисел, вводимых с клавиатуры. Необходимо убедиться, что программа правильно

работает при граничных условиях. Это пары чисел: 00; 09; 0A; 0F; 90; A0; F0. Используйте точку останова для запуска программы без выполнения инструкции INT 20h. (Для ввода шестнадцатеричных чисел используйте только заглавные буквы.)

Процедуры

Процедура – это список инструкций, который можно вызывать из различных мест нашей программы.

Для вызова процедуры будем использовать инструкцию CALL 200h, где 200h – адрес, по которому находится первая инструкция процедуры. Вспомним, что первая инструкция нашей программы находится по адресу l00h. Располагая процедуру по адресу 200h, мы стремимся убрать ее подальше от основной программы. Если мы запишем список инструкций процедуры (тело процедуры) по другому адресу, то этот адрес следует записать вместо адреса 200h.

Следующая программа вызывает десять раз процедуру, которая каждый раз печатает один символ, начиная от A и заканчивая J:

```
100     MOV     DL, 41
102     MOV     CX, 000A
105     CALL    0200
108     LOOP    0105
10A     INT     20
```

Текст процедуры:

```
200     MOV     AH, 02
202     INT     21
204     INC     DL
206     RET
```

В тексте процедуры содержатся две новые инструкции. Инструкция INC увеличивает содержимое регистра DL на единицу. Инструкция RET возвращает управление в то место основной программы, откуда процедура была вызвана. Следующей инструкцией, выполняемой после инструкции RET, будет та инструкция основной программы, которая следует за инструкцией CALL. В примере это инструкция LOOP.

В в е д и т е основную программу и процедуру в память. С помощью команды G выполните программу, а затем протрассируйте ее с целью детального изучения работы. При этом особое внимание уделите изменению регистра IP.

Работа со стеком

Стек – это любая область ОП, при работе с которой допустимы только две операции: 1) добавление элемента данных (например, слова) в вершину стека; 2) выборка элемента данных из вершины стека. От других операций

над областью памяти, занимаемой стеком, мы добровольно отказываемся. Добавление слова в стек выполняет инструкция PUSH , а выборку слова из стека – инструкция POP.

При работе со стеком выполняется правило “последним пришел, первым ушел". Это аналогично стопке подносов в столовой: поднос, который был положен на стопку последним, первым будет с нее снят.

Где находится стек в ОП? Для того, чтобы выполнять операции со стеком, достаточно знать адрес в памяти вершины стека. Этот адрес всегда содержится в паре регистров: 1) регистр сегмента стека – SS; 2) указатель стека – SP (от "Stack Pointer" – указатель стека). Реальный адрес ячейки, являющейся вершиной стека, получается аппаратно путем суммирования содержимого SS, умноженного на 16, с содержимым SP.

Содержимое регистров SS и SP вы многократно наблюдали ранее, получая листинг регистров по команде R или какой-то другой команде Debug. Первоначальную запись в данные регистры выполняет DOS, руководствуясь для .com-программ следующим. Во-первых, в SS записывается тот же номер параграфа, что и в регистр сегмента кодов CS. Это означает, что для стека отводится тот же сегмент ОП, что и для программы (рис. 6).

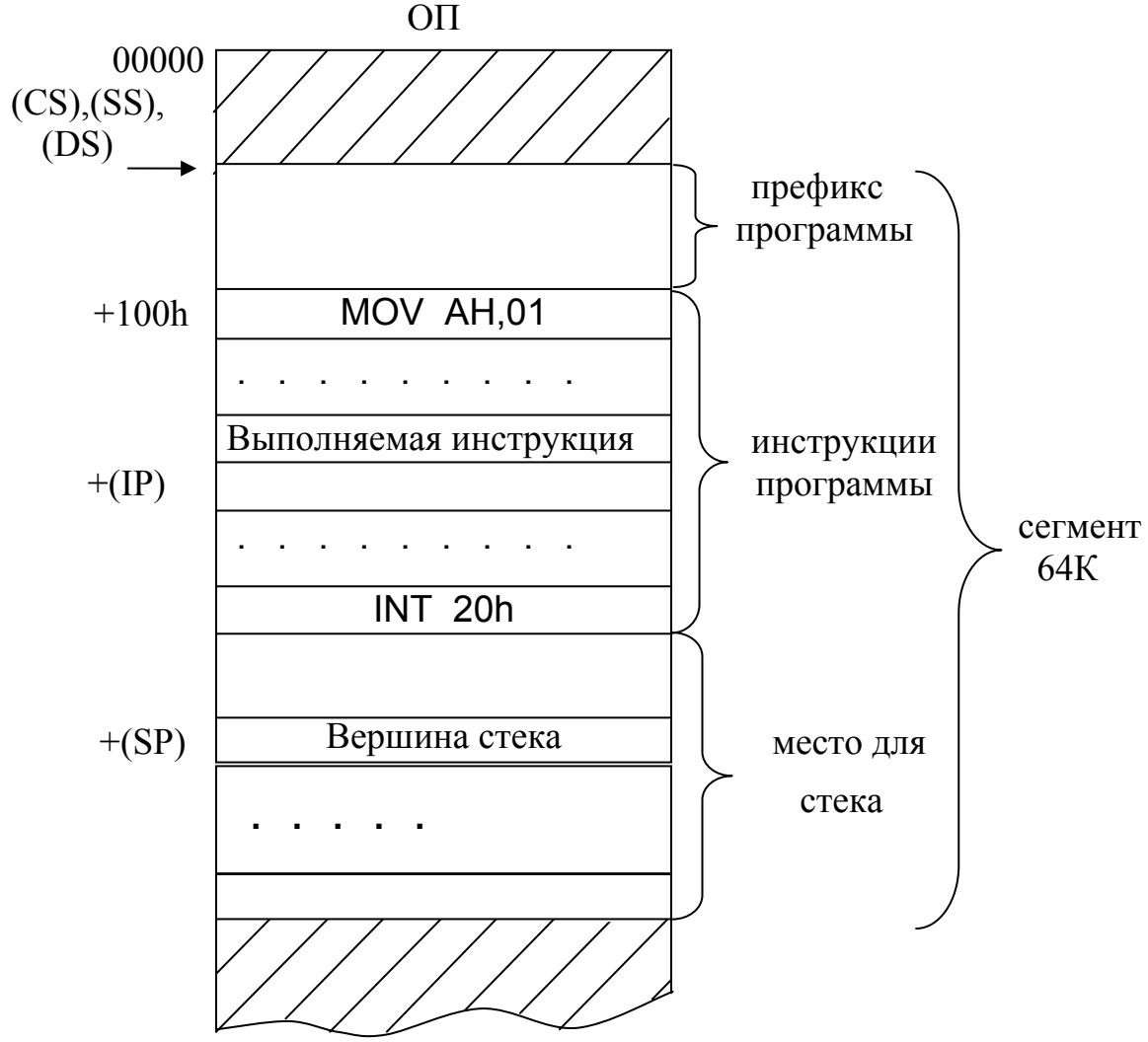


Рис. 6. Размещение в памяти односегментной программы

В указатель стека DOS записывает максимально возможное число, в результате чего вершиной стека первоначально является последняя ячейка сегмента программы (и сегмента стека). "Растет" стек в отличие от программы в сторону не больших, а меньших адресов. ***Наберите*** команду R и внимательно посмотрите на содержимое SS и SP.

Стек широко используется для временного хранения данных. Мы рассмотрим два его применения. Первое из них связано с хранением адреса возврата при выполнении процедуры. В самом деле, при выполнении инструкции вызова процедуры CALL мы "перескакиваем" в другую область ОП (например, по адресу 200h), где находится первая инструкция тела вызываемой процедуры. Так как при выполнении инструкции процедуры RET мы возвращаемся в основную программу, то необходимо где-то хранить адрес возврата (адрес инструкции, следующей за инструкцией CALL). В качестве места хранения адреса возврата и используется программный стек.

Вызов процедуры не требует от нас использования инструкций PUSH и POP, т.к. работа со стеком в данном случае производится "автоматически" – инструкция CALL помещает адрес возврата в стек, а инструкция RET извлекает его оттуда.

Введите в память (если они там не сохранились) предыдущие программу и процедуру. Протестировав программу, наблюдайте за содержимым указателя стека SP до и после исполнения инструкций CALL и RET. Сразу же после любого выполнения инструкции CALL найдите в стеке адрес возврата (108), используя команду U Debug.

Полезность стека становится очевидной при использовании в программе вложенных процедур, когда одна вызываемая процедура вызывает другую процедуру, которая, в свою очередь, вызывает третью и т.д. При этом порядок записи адресов возврата как раз и должен соответствовать правилу "последним пришел, первым ушел".

Допустим, что программа состоит из главной программы и трех процедур. Главная программа имеет вид:

```
100     CALL    200
103     INT     20
```

По адресу 200 находится процедура, которая печатает букву A и вызывает вторую процедуру, записанную по адресу 300, которая выводит B. Вторая процедура вызывает третью процедуру, первая инструкция которой находится по адресу 400. Эта третья процедура выполняет только вывод на экран буквы C.

Введите данную программу в память и протрассируйте ее,

наблюдая за изменением регистра SP и содержанием стека. Обязательно найдите в стеке адреса возврата, когда вызваны все три процедуры.

Другое применение стека: сохранение регистров в начале процедуры и восстановление их содержимого в конце процедуры. Пример процедуры, в которой производится такое сохранение:

```
200     PUSH    CX
201     PUSH    DX
202     MOV     DL, 0A
205     CALL    300
208     INC     DL
20C     POP     DX
20D     POP     CX
20E     RET
```

Обратите внимание, что инструкции POP расположены по отношению к инструкциям PUSH в обратном порядке, так как POP удаляет слово, помещенное в стек последним, а старое значение CX находится в стеке "глубже" старого значения DX.

Сохранение и восстановление регистров CX и DX позволяет произвольно изменять их значения внутри процедуры (которая начинается с адреса 200h), в то же время значения регистров, используемых процедурами, вызывающими эту, сохранены. Следовательно, мы можем использовать эти регистры для хранения локальных переменных – переменных, которые применяются внутри процедур, не влияя на значения переменных, используемых вызывающей программой.

В дальнейшем обязательно выполняйте правило: все регистры, содержимое которых меняется внутри процедуры, за исключением регистров, используемых для передачи выходных данных процедуры в вызвавшую ее программу, в конце процедуры должны восстанавливать свои прежние значения.

Более совершенный ввод шестнадцатеричных цифр

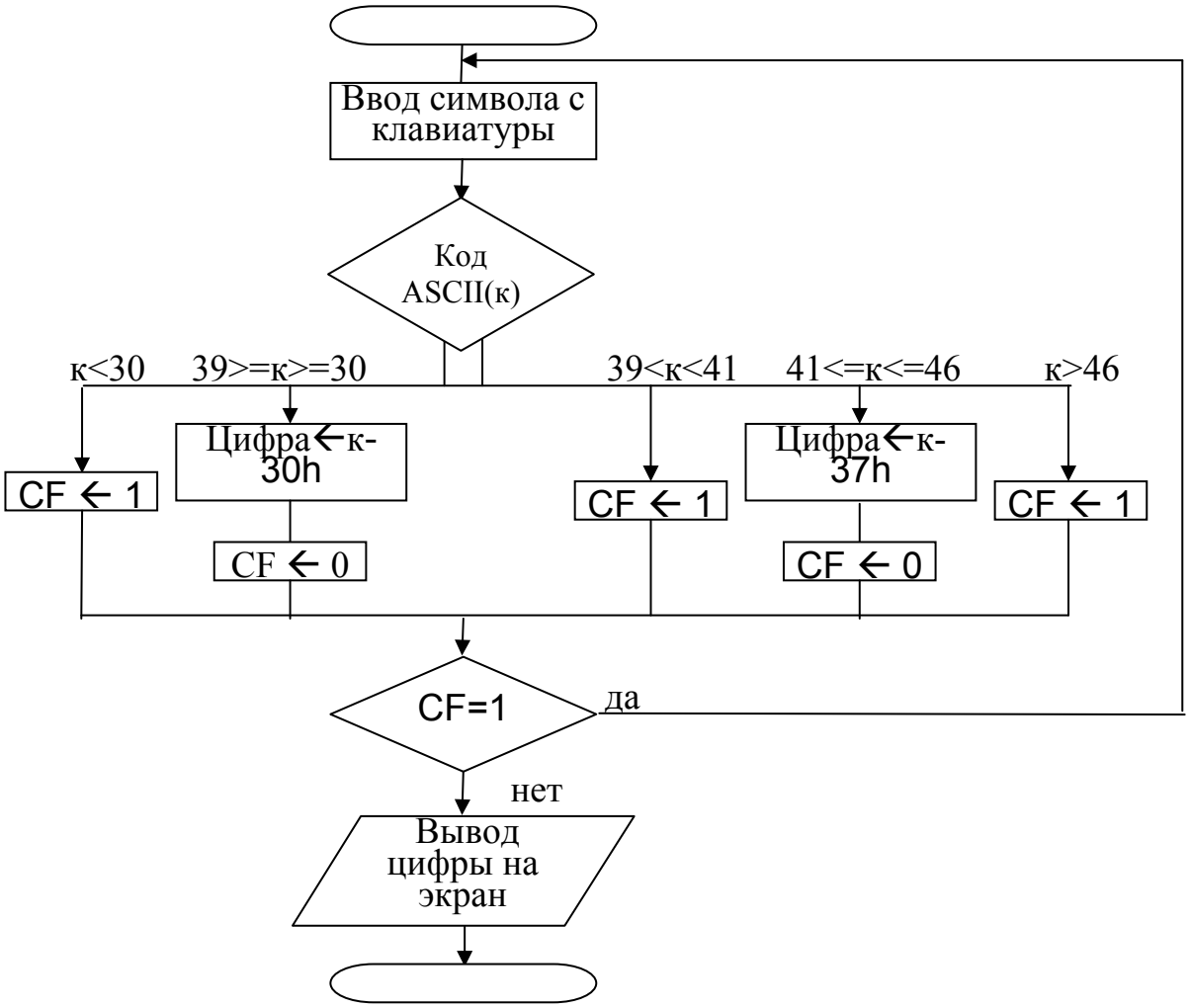
Записанные ранее программы ввода одно- и двузначного шестнадцатеричных чисел реагировали на нажатие “нецифровых” клавиш точно также, как и “цифровых”. Теперь мы получим программу, которая не будет реагировать на нажатие “нецифровых” клавиш.

Для этого в состав программы введем новую процедуру ввода шестнадцатеричной цифры. Эта процедура возвратит управление в главную программу только тогда, когда она получит с клавиатуры правильную шестнадцатеричную цифру. При нажатии "нецифровой" клавиши ее код на экран не выводится и управление в программу не возвращается.

Ранее для ввода символа с клавиатуры использовалась функция

1 программного прерывания 21h. Эта функция не только вводит символ с клавиатуры, но и выводит его на экран. Подобный вывод символа во время его ввода называется "эхом" символа. Теперь мы будем использовать функцию 8 21-го прерывания, которая не выводит "эхо" символа.

На рис.7 приведена блок-схема новой процедуры ввода шестнадцатеричной цифры, а на рис. 8 – сама эта процедура. (Процедура возвращает цифру в регистре BL.) ***И з у ч и т е*** внимательно алгоритм и найдите реализацию его этапов в программе. Во-первых, обратите внимание, что управляющей структурой верхнего уровня является цепочка из цикла ПОКА и этапа "Вывод цифры на экран". В качестве условия повторения цикла выступает равенство единице флага ошибки CF.



CF - флаг ошибки (0 - ошибки нет, 1 - ошибка есть)

Рис. 7. Алгоритм ввода одной шестнадцатеричной цифры

Имя CF флага ошибки обусловлено тем, что он реализуется в программе с помощью одноименного флага переноса. При этом флаг переноса используется совсем не для того, для чего он был создан. Данный прием

широко распространен, и мы также будем его использовать.

Существуют специальные инструкции для работы с флагом CF. Инструкция STC устанавливает флаг (CF=1), а CLC сбрасывает его (CF=0). Инструкция условного перехода JC выполняет переход при CF=1, а инструкция JNC - при CF=0.

В в е д и т е данную процедуру в память. Кроме того, запишите в память следующую программу для проверки процедуры:

```
100     CALL    200
103     INT     20
```

П р о т р а с с и р у й т е программу, используя команду P для перехода через инструкции INT. Курсор появится в левой части экрана и будет ждать ввода символа. Напечатайте символ "К", который не является правильным символом. Ничего не произойдет. Теперь напечатайте один из заглавных шестнадцатеричных символов. Вы должны увидеть шестнадцатеричную цифру в BL, а также на экране. Испытайте эту процедуру на граничных условиях: "\" (символ, стоящий перед нулем), "0", "9", ":" (символ, стоящий после 9), и т.д.

```
200     PUSH    DX
201     MOV     AH, 8
203     INT     21
205     MOV     DL, AL
207     CMP     AL, 30
209     JB      221
20B     CMP     AL, 39
20D     JA      214
20F     SUB     AL, 30
211     CLC
212     JMP     222
214     CMP     AL, 41
216     JB      221
218     CMP     AL, 46
21A     JA      221
21C     SUB     AL, 37
21E     CLC
21F     JMP     222
221     STC
222     JC      203
224     MOV     BL, AL
226     MOV     AH, 2
228     INT     21
22A     POP     DX
22B     RET
```

Рис. 8. Процедура ввода шестнадцатеричной цифры

Теперь, когда у нас есть процедура ввода одной шестнадцатеричной цифры, программа, считывающая двузначное шестнадцатеричное число в регистр DL и обрабатывающая ошибки, стала достаточно простой. Ее алгоритм приведен на рис. 9.

Рис. 9. Алгоритм ввода двузначного шестнадцатеричного числа

Задание

Р а з р а б о т а й т е программу, вызываемую из DOS, которая выполняет:

1) ввод с клавиатуры двух 4-х значных шестнадцатеричных чисел, которые записываются в качестве содержимого регистров BP и DI;
2) вывод на экран содержимого регистров, заполненных на шаге 1, в виде двоичных чисел;
3) вывод на экран содержимого регистров, заполненных на шаге 1, в виде шестнадцатеричных чисел.

Пример информации на экране:

ВВЕДИТЕ СОДЕРЖИМОЕ РЕГИСТРА BP F46B<Enter>
ВВЕДИТЕ СОДЕРЖИМОЕ РЕГИСТРА DI 5A0C<Enter>

(BP) = 1111010001101011 (DI) = 0101101000001100
(BP) = F46B (DI) = 5A0C

Примечание 1. В отличие от приводимых в описании работы программ, некоторые 8-битные регистры следует заменить на 16-битные.

Примечание 2. Рекомендуется дополнительно разработать процедуру, выполняющую ввод шестнадцатеричного числа в 16-битный регистр, процедуру вывода содержимого такого регистра в двоичном виде, а также процедуру вывода содержимого 16-битного регистра в шестнадцатеричном виде.

Примечание 3. При реализации вывода второй и третьей шестнадцатеричных цифр числа, сдвигу числа вправо должен предшествовать его сдвиг влево. Для выполнения сдвига влево используйте инструкцию SHL ("Shift Left" – логический сдвиг влево). Использование этой инструкции аналогично SHR. Выполнение SHL имеет такой же эффект, как и умножение на два, четыре, восемь и так далее, в зависимости от числа (соответственно единицы, двойки или тройки), хранящегося в CL.

Примечание 4. Для получения на экране достаточно хорошей формы представления информации выполняйте вывод промежуточных пробелов. Число пробелов определяйте опытным путем.

Отлаженную .com-программу занесите для пересылки на дискету в каталог LAB2.

Лабораторная работа N 3

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ

Цель работы

До сих пор нашим единственным помощником при написании и отладке машинных программ была системная программа Debug. Мы и далее будем широко использовать Debug при отладке своих программ. Что касается написания программы, то тут помощь Debug явно недостаточна, и процесс написания сколько-нибудь сложной программы скорее всего продлится очень долго. По этой причине мы переходим к написанию программ на языке ассемблера.

Целью выполнения данной работы является получение начальных навыков по разработке программ на языке ассемблера. А именно – рассматриваются псевдооператоры, позволяющие разрабатывать простые ассемблерные программы, а также производится первоначальное

знакомство с системными программами (EDIT, TASM и TLINK), обеспечивающими преобразование программы на языке ассемблера в машинную программу.

Общая структура простых ассемблерных программ

Основным отличием программы на языке ассемблера от соответствующей машинной программы, которую мы вводим в ЭВМ с помощью Debug, является наличие псевдооператоров. В отличие от исполнительного оператора, который преобразуется транслятором-ассемблером в одну машинную инструкцию, псевдооператор ни в какие машинные команды не транслируется. Такой оператор представляет собой указание транслятору и нигде, кроме самого транслятора, не используется.

Общая структура простых ассемблерных программ, которая будет для нас достаточна на ближайшее время, имеет вид:

```
Code_seg      SEGMENT
                ASSUME    CS:Code_seg
                ORG    l00h
Глав_прос       PROC   NEAR
                 . . . . . . . . .
                INT     20h
Глав_прос       ENDP
Code_seg       ENDS
              END  Глав_прос
```

Посмотрим внимательно на данную структуру. По горизонтали она разделена на две части. В левой части записаны идентификаторы (метки) программных объектов, к которым относятся сегменты, процедуры, исполнительные операторы и элементы данных. В правой части листинга находятся псевдооператоры и исполнительные операторы, а также операнды этих операторов. Идентификатор отделяется от соответствующего оператора минимум одним пробелом.

Простейшая программа состоит всего из одного сегмента кодов, в котором находятся инструкции программы. Начало данного сегмента обозначено в программе с помощью псевдооператора SEGMENT. Для обозначения данного сегмента в программе используется имя Code_seg, но вы можете применить для этого любое другое имя (правило написания имен будет рассмотрено немного позже). Псевдооператору SEGMENT всегда соответствует псевдооператор ENDS, обозначающий конец сегмента. Оба псевдооператора имеют один и тот же идентификатор (например, Code_seg).

Следующим после псевдооператора SEGMENT в программе записан псевдооператор ASSUME, с помощью которого вы информируете

транслятор о том, что в регистре CS будет находиться начальный адрес (а точнее – начальный номер параграфа) сегмента программы. Что касается записи в регистр CS, то ее транслятор не делает. (Данная запись выполняется много позже, перед самым выполнением программы. Это делает операционная система (а точнее – загрузчик). Фактически запись в регистры CS и IP означает передачу управления загруженной программе.)

Следующий псевдооператор ORG l00h сообщает транслятору о том, что самый первый исполнительный оператор нашей программы должен быть помещен в выделенный программе сегмент ОП со смещением l00h относительно начала сегмента. Мы и раньше использовали это смещение, вводя машинные программы с помощью Debug. Следует обратить внимание на символ h после шестнадцатеричного числа 100. Использование этого символа после шестнадцатеричных чисел обязательно. Так как транслятор-ассемблер в отличие от Debug "обычной" считает не шестнадцатеричную, а десятичную систему счисления.

Следующий псевдооператор PROC NEAR информирует транслятор о том, что далее начинаются исполнительные операторы (инструкции) основной программы. Вид данного псевдооператора обусловлен тем, что наша основная программа рассматривается операционной системой как подпрограмма (процедура). Операнд NEAR ("близкий") означает, что процедура не вызывается извне данного сегмента. После завершения тела процедуры записывается псевдооператор ENDP, которому предшествует имя процедуры.

Если в нашей программе есть не только главная, но и другие процедуры, то их тексты записываются после главной процедуры и оформляются такими же псевдооператорами PROC и ENDP.

В конце текста программы на ассемблере записывается псевдооператор END, который информирует транслятор о том, что текст программы закончился. Операнд псевдооператора END есть адрес той инструкции программы, которая должна выполняться первой. Так как мы желаем, чтобы первой выполнилась самая первая инструкция программы, то и записали в качестве операнда для END имя основной программы.

Пример программы на ассемблере

В качестве примера запишем на ассемблере ту программу Writestr, которую мы создали в работе 1 с помощью Debug. Напомним текст машинной программы:

```
100     MOV     AH, 02
102     MOV     DL, 2A
104     INT     21
106     INT     20
```

Соответствующий текст программы на ассемблере:

```
Code_seg      SEGMENT
              ASSUME   CS:Code_seg
                 ORG    l00h
 Writestr       PROC      NEAR
                  MOV      AH, 2h
                  MOV      DL, 2Ah
                  INT       21h
                  INT       20h
 Writestr       ENDP
  Code_seg    ENDS
          END    Writestr
```

Следует отметить, что если в программе есть числа типа ACh, то чтобы транслятор не запутался с ними (он может принять их за имя), всякое шестнадцатеричное число, начинающееся с буквы, следует предварять нулем. Например: 0ACh.

Подготовка программы к выполнению

Текст программы на ассемблере называется исходной программой. Для того чтобы преобразовать этот текст в машинную программу, нам нужна помощь не только со стороны транслятора, но и от некоторых других системных программ.

Во-первых, с помощью текстового редактора мы получаем исходный файл программы. Имя исходного файла должно иметь расширение .asm. Можно использовать любой редактор, который выдает результирующий текст в коде ASCII. Примером такого редактора является Edit. При использовании Commander работа с Edit заключается в следующем.

При создании нового файла необходимо одновременно нажать клавиши <Shift> и <F4>. В появившемся на экране окне необходимо набрать имя создаваемого файла (например, Writestr.asm) и нажать клавишу <Enter>. В ответ на последующий вопрос также нажимается эта клавиша. Для работы с ранее созданным текстовым файлом необходимо установить псевдокурсор в каталоге файлов на нужный файл и нажать клавишу <F4>. Для того, чтобы переписать откорректированный файл из ОП на диск, необходимо нажать <F2>.

С о з д а й т е исходный файл Writestr.asm. Убедитесь, что это именно ASCII-файл. Для этого, находясь в DOS, напечатайте:

C > TYPE Writestr.asm

Вы должны увидеть тот же текст, который ввели в текстовом редакторе. Если вы увидите в вашей программе странные символы, то для ввода текста программ следует использовать другой текстовый редактор. Кроме того, в файле необходимо оставлять пустую строку после оператора END. Теперь давайте начнем ассемблировать программу Writestr:

```
C > TASM   Writestr
```

Ответное сообщение транслятора будет:

```
Turbo Assembler Version 3.1  Copyright(c) 1988
Assembling file:        WRITESTR.ASM
Errог messages:         None
Warning messages:       None
Passes:                 1
Remaining memory:       427K
C >
```

В результате транслятор-ассемблер создал файл, называющийся Writestr.obj, который вы найдете на диске. Это промежуточный файл, называющийся объектным файлом. Он содержит программу на машинном языке, вместе с большим количеством вспомогательной информации, используемой другой программой, называющейся редактором связей или компоновщиком. В качестве такого редактора связей мы будем использовать Tlink (Turbo Link).

Tlink преобразует объектный файл в загрузочный модуль – файл с расширением .exe или файл с расширением .com. Для получения файла типа .com при вызове Tlink необходимо задать дополнительный параметр t. ***П о п р о с и т е*** редактор связей получить .com-файл для нашей программы Writestr:

```
C > TLINK   Writestr/t
```

Чтобы проверить результат работы программы Tlink, посмотрим на файлы с именем Writestr, которые мы создали:

```
C > DIR   Writestr.*
```

Ответ системы:

```
 Volume in drive C has no label
 Directory  of  C:\
 WRITESTR  ASM        192      1-01-90      12:07a
 WRITESTR  MAP         99      1-01-90       2:11a
 WRITESTR  OBJ        147      1-01-90       2:10a
 WRITESTR  COM          8      1-01-90       2:11a
     4  File (s)  18403328  bytes  free
 C >
```

Это точное число файлов, включая Writestr.com. Файл с расширением .map создается редактором связей одновременно с файлом загрузочного модуля. Назначение этого файла будет понятно из следующей лабораторной работы.

Напечатайте "Writestr", чтобы запустить .com-версию и убедитесь, что Ваша программа функционирует правильно (напоминаем, что она должна печатать звездочку на экране).

Теперь введем созданный .com-файл в Debug и разассемблируем его, чтобы увидеть получившуюся машинную программу:

```
C > DEBUG    Writestr.com
_U
1593:0100       B402        MOV       AH, 02
1593:0102       B22A        MOV       DL, 2A
1593:0104       CD21        INT       21
1593:0106       CD20        INT       20
```

Получили именно то, что мы уже имели в работе 2.

Комментарии

Программа на ассемблере может содержать любые сообщения, информирующие программиста о содержании текста программы. Такое сообщение называется комментарием. Комментарии могут находиться на любых строках программы. На каждой строке, где есть комментарий, ему предшествует точка с запятой (;). Подобно псевдооператорам комментарии не транслируются ни в какие машинные инструкции. Но в отличие от псевдооператоров, они не интересуют и сам транслятор, существуя лишь для человека, который читает исходную программу.

Добавим комментарии в записанную ранее программу на ассемблере:

```
Code_seg    SEGMENT
                    ASSUME    CS:Code_seg
                    ORG       100h
;
;              Вывод звездочки на экран
;              ---------------------------------
;
Writestr    PROC    NEAR
              MOV     AH, 2h        ; Функция вывода символа
              MOV     DL, 2Ah       ;  Символ * в DL
              INT     21h           ;  Вывод символа
              INT     20h           ;  Возврат в DOS
Writestr    ENDP
Code_seg    ENDS
                    END       Writestr
```

Теперь можно легко понять смысл программы. Обратите внимание на комментирование процедуры. Каждая процедура должна обязательно иметь

вводные комментарии, которые содержат:

1) словесное описание функций процедуры;
2) перечень и способ передачи каждого входного и выходного параметра процедуры;
3) перечень процедур, вызываемых в данной процедуре;
4) перечень переменных (областей памяти), которые используются процедурой, с указанием для каждой переменной способа ее использования. Допустимые способы использования: чтение; запись; чтение-запись.

Для приведенной выше простой процедуры вводный комментарий содержит лишь описание функции процедуры.

Что касается текущих комментариев, поясняющих назначение отдельных операторов программы и их групп, то к ним нет жестких требований. Желательно, чтобы были прокомментированы основные управляющие структуры программы (цепочки, ветвления и циклы). Кроме того, должны быть прокомментированы вызовы подпрограмм, т.е. операторы CALL и INT. Каждый такой оператор инициирует десятки или сотни машинных инструкций и поэтому заслуживает пояснения.

Следует помнить, что исходная программа без комментариев не имеет какой-либо коммерческой ценности. Это просто напросто черновик автора программы. Более того, по истечению нескольких месяцев даже автору требуются значительные усилия на то, чтобы понять смысл программы.

Метки

При построении нами машинных программ с помощью Debug инструкции переходов содержали адреса тех инструкций программы, на которые переход делался. Допустим, что мы решили добавить в программу новые инструкции. В этом случае от нас требуется изменить многие ранее записанные адреса переходов, что чрезвычайно неудобно.

При написании программы на ассемблере вместо адресов перехода мы используем метки тех операторов, на которые делается переход. То же самое относится и к процедурам: первые операторы процедур мы помечаем метками, которые теперь являются именами соответствующих процедур и используются в операторах вызова этих процедур. Рассмотрим правило записи меток.

Максимальная длина метки – 31 символ. Это могут быть следующие символы:

1) буквы – от A до Z и от a до z;
2) цифры – от 0 до 9;
3) специальные символы – знак вопроса (?) ; знак @ ; точка (.) ; подчеркивание (_) ; доллар ($) .

Первым символом в метке должна быть буква или специальный символ. Точка может использоваться в метке только в качестве первого символа.

Ассемблер не делает разницы между заглавными и строчными буквами. Примеры меток: count, Page25, $E10. Весьма желательно, чтобы метки поясняли смысл программы, а не выбирались отвлеченно.

Некоторые из буквенных слов зарезервированы транслятором-ассемблером и не могут быть использованы в качестве меток. Сюда относятся имена регистров (например, AX), мнемоники исполнительных инструкций (например, ADD) и псевдооператоры (например, END).

Еще один пример программы

Ранее в работе 2 мы создали небольшую программу, которая печатала буквы от A до J. Вот эта машинная программа:

```
100     MOV     DL, 41
102     MOV     CX, 000A
105     CALL    0200
108     LOOP    0105
10A     INT     20
200     MOV     AH, 02
202     INT     21
204     INC     DL
206     RET
```

Перепишем теперь эту программу на языке ассемблера:

```
        Code_seg        SEGMENT
                        ASSUME          CS:Code_seg
                        ORG     l00h
;
;               Вывод 10 символов от A до J
;               -----------------------------------
;   Вызовы:     Write_char
;
Print_a_j       PROC    NEAR
                        MOV     DL, 'A'         ; В DL код буквы A
                        MOV     CX, 10          ; В счетчике символов 10
  Print_loop:           CALL    Write_char      ; Вывод символа
                        LOOP    Print_loop      ; Переход к следующему
                                                ; символу
                        INT     20h             ; Возврат в DOS
Print_a_j       ENDP
;
;       Вывод символа на экран и увеличение кода символа на 1
;       ------------------------------------------------------------------
```

```
;   Входы:     DL содержит код символа
;   Выходы:   DL содержит код нового символа
;
Write_char    PROC     NEAR
                  MOV        AH, 02      ; Функция вывода символа
                  INT        21h          ; Вывод символа на экран
                  INC        DL           ; Код следующего символа
                  RET                     ; Возврат из процедуры
Write_char    ENDP
;
Code_seg      ENDS
                  END     Print_a_j
```

Обратите внимание, что в первой инструкции процедуры Print_a_j второй операнд представляет собой символ, заключенный в кавычки (кавычки могут быть как одиночными, так и двойными). Это означает, что операнд представляет собой код ASCII-символа, заключенного в кавычки.

Отметим попутно еще одну весьма полезную деталь ассемблера. Данный язык позволяет записывать в качестве операнда арифметическое выражение, элементами которого являются константы. Например, инструкция MOV DL,‘A’-10 помещает в регистр DL код ASCII-символа A, уменьшенный на 10.

В в е д и т е эту программу в файл Printaj.asm и получите файл Printaj.com. Затем выполните программу. Добившись правильной ее работы, используйте Debug для разассемблирования программы. Посмотрите, как транслятор распределил память для основной программы и для процедуры. Если мы раньше перестраховывались, располагая процедуру подальше (по адресу 200h), то теперь транслятор расходует память экономно, не оставляя промежутков между процедурами.

Псевдооператоры определения данных

Обычно программа использует оперативную память не только для хранения своих инструкций, но и для хранения данных, обрабатываемых этими инструкциями. Например, одна из программ в работе 1 выводила на экран текстовое сообщение “Hello, DOS here”. Мы сами выбрали область памяти и записали в нее требуемое сообщение, используя команды DEBUG. Теперь такие действия мы поручим транслятору, записав в программу на ассемблере псевдооператор:

```
Text    DB      ‘Hello, DOS here’, 24h
```

Который просит транслятор зарезервировать область памяти длиной 16 байт, записать в нее коды ASCII символов, заключенных в кавычки, поместить в последний байт код 24h, а также обозначить меткой Text адрес

первого байта этой области (этот байт содержит код буквы H). Аналогичный результат обеспечивает псевдооператор:

```
Text    DB      'Hello, DOS here$'
```

Если нас не интересует первоначальное содержимое резервируемого байта памяти, то мы задаем этот байт символом "?". Например, следующий псевдооператор резервирует без инициализации 10 байт памяти:

```
F2db    DB      ?,?,?,?,?,?,?,?,?,?
```

Это же делает и псевдооператор:

```
F3db    DB      10  DUP  (?)
```

Что касается размещения псевдооператоров определения данных в исходной программе, то оно должно гарантировать непопадание на процессор соответствующего поля данных в качестве исполняемой машинной инструкции. Это можно обеспечить двумя способами. Во-первых, эти псевдооператоры можно записать в самом начале программы, предусмотрев "перепрыгивание" через них (а точнее – через соответствующие данные) с помощью оператора безусловного перехода JMP. Во-вторых, их можно поместить в конце программы, после оператора INT 20h , или после оператора RET. С учетом этого запишем текст программы, выводящей на экран "Hello, DOS here":

```
Code_seg    SEGMENT
                    ASSUME    CS:Code_seg
                    ORG       100h
;
;             Вывод строки на экран
;             ----------------------------
;
Print_he    PROC    NEAR
              MOV     AH, 9h                ; Функция вывода строки
              LEA     DX, Text              ; DX ← адрес строки
              INT     21h                   ; Вывод строки
              INT     20h                   ; Возврат в DOS
 Text         DB     'Hello, DOS here', 24h ; Строка символов
Print_he    ENDP
Code_seg    ENDS
                    END   Print_he
```

Инструкция LEA выполняет загрузку адреса (смещения относительно начала сегмента) второго операнда в качестве содержимого первого операнда.

В в е д и т е эту программу в файл Printhe.asm и получите файл Printhe.com. Затем выполните программу.

Вывод на экран двузначного шестнадцатеричного числа

В работе 2 подобная программа была разработана нами с помощью Debug. Теперь мы запишем ее на языке ассемблера, добавив небольшую процедуру, которая выводит один символ на экран. Выделение этой, на первый взгляд лишней процедуры, обусловлено желанием в дальнейшем вносить изменения в операцию вывода символа, совершенно не влияя на остальную часть программы.

На рис. 10 приведено дерево подпрограмм для данной программы. Эта структура показывает, какие подпрограммы (процедуры) входят в состав программы, и как эти процедуры связаны между собой по управлению. При этом самой верхней (т.е. главной процедурой) является Test_write_hex. Данная процедура предназначена для тестирования процедуры Write_hex, выполняющей вывод на экран двузначного шестнадцатеричного числа. В свою очередь, процедура Write_hex вызывает (инициирует) процедуру Write_hex_digit, выполняющую вывод на экран шестнадцатеричной цифры. В ходе своей работы данная процедура вызывает процедуру Write_char, выполняющую вывод символа на экран.

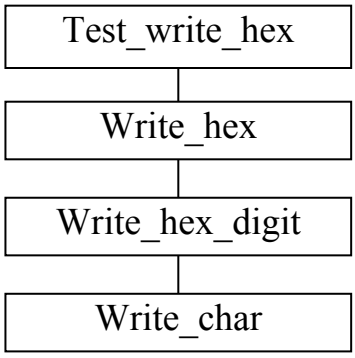


Рис. 10. Дерево подпрограмм для программы вывода двузначного шестнадцатеричного числа

В работе 2 приведены блок-схемы для процедур Write_hex и Write_hex_digit. В следующей программе тексты этих процедур опущены:

```
Code_seg        SEGMENT
                ASSUME  CS:Code_seg
                ORG     l00h
;
;           Тестирование процедуры  Write_hex
;           ------------------------------------------------
;   Вызовы:   Write_hex
;
Test_write_hex      PROC   NEAR
```

```
                    MOV     DL, 3Fh.        ; Тестировать с 3Fh
                    CALL    Write_hex       ; Вывод числа
                    INT         20h         ; Возврат в DOS
Test_write_hex      ENDP
;
;           Вывод  двузначного шестнадцатеричного числа
;           ------------------------------------------------------
;   Входы:  DL  содержит  выводимое  число
;   Вызовы:   Write_hex_digit
;
Write_hex       PROC        NEAR
  .   .   .   .   .   .   .   .
Write_hex       ENDP
;
;                   Вывод шестнадцатеричной цифры
;                   ----------------------------------------
;    Входы:   BL содержит выводимую цифру
;    Вызовы:  Write_char
;
Write_hex_digit   PROC   NEAR
  .   .   .   .   .   .   .   .
Write_hex_digit   ENDP
;
;                 Вывод символа на экран
;                 ------------------------------
;   Входы:  DL  содержит  байт,  выводимый  на  экран
;
Write_char      PROC        NEAR
                  PUSH      AX
                  MOV       AH, 2       ; Функция вывода символа
                   INT      21h         ; Вывод символа на экран
                   POP      AX
                   RET
Write_char      ENDP
;
Code_seg        ENDS
  END   Test_write_hex
```

З а п и ш и т е данную программу полностью и поместите ее в файл Video_io.asm. Получите файл Video_io.com и используя Debug тщательно протестируйте программу, меняя 3Fh по адресу 101h на каждое из граничных условий, которые использовались ранее в работе 2 для проверки программы, выполняющей те же функции.

Обязательно сохраните файл Video_io.asm . В дальнейшем мы будем добавлять в него новые процедуры. Пользуясь этими процедурами как "кирпичиками", мы сможем создавать достаточно сложные программы.

Примечание. Для отладки Вашей программы удобно использовать подход, называемый "снизу-вверх". Согласно ему сначала отлаживаются процедуры, расположенные внизу дерева подпрограмм. После того, как эти процедуры отлажены, отлаживаются процедуры их вызывающие и т.д.

Реализация данного подхода предполагает первоначальную корректировку процедуры Test_write_hex так, чтобы из нее вызывалась не процедура Write_hex, а процедура Write_hex_digit. После того, как Write_hex_digit отлажена, Test_write_hex восстанавливается и программа отлаживается целиком.

Алгоритм вывода на экран десятичных чисел

Содержимое ячейки памяти (байта или слова) может быть выведено на экран не только в виде двоичного или шестнадцатеричного числа, но и в любой другой системе счисления, например, в десятичной.

Назовем процедуру, выполняющую вывод на экран десятичного представления слова данных, содержащего двоичное число без знака, как Write_dec. Пусть это слово данных передается на вход процедуры в регистре DX. Обсудим алгоритм процедуры Write_dec.

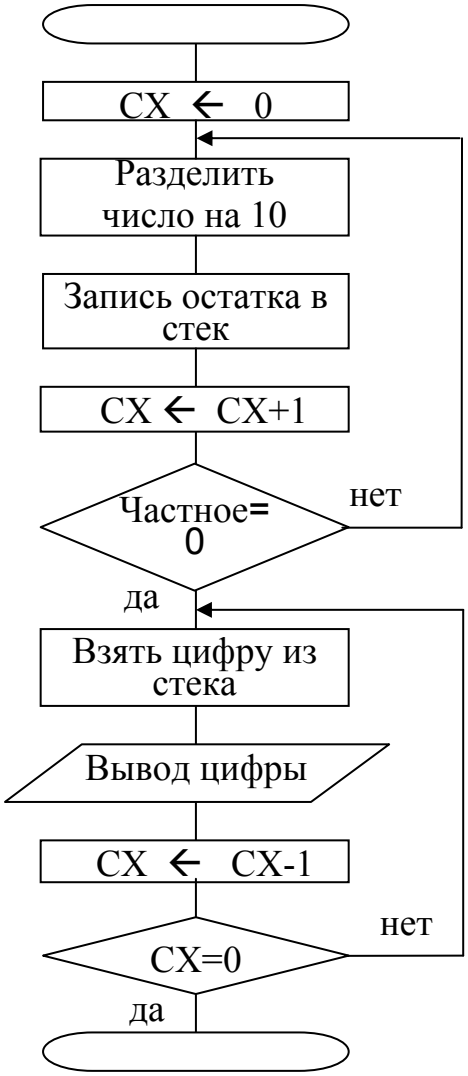
Предполагаемый результат работы процедуры состоит в том, что сначала на экран будет выведена старшая десятичная цифра числа, затем вторая слева цифра и т.д. Для этого следует иметь десятичное представление числа, хранящегося у нас в двоичном виде. Вспомним, что при делении числа на 10 остаток есть младшая десятичная цифра числа. Если полученное частное разделим на 10, то получим вторую справа десятичную цифру. Подобное деление можно продолжать до тех пор, пока очередное частное не будет меньше 10 и, следовательно, оно и будет равно старшей цифре числа. Если бы порядок получения десятичных цифр из числа совпал бы с порядком их вывода на экран, то задача была бы уже решена. Но у нас эти порядки противоположны. Поэтому мы опять обратимся за помощью к стеку. Вспомним его свойство "кто пришел первым, тот уйдет последним". Поэтому, если мы поместим в стек младшую десятичную цифру числа (она получена первой), то она будет извлечена из стека для вывода на экран последней. Блок-схема соответствующего алгоритма приведена на рис. 11.

Дерево подпрограмм

Кроме самой процедуры Write_dec, выполняющей вывод на экран десятичного представления слова данных, наша программа включает и

другие процедуры (рис.12). Во-первых, это главная процедура Test_wrte_dec, используемая для тестирования Write_dec. Во-вторых, это процедура вывода шестнадцатеричной цифры Write_hex_digit (она

пригодна и для вывода десятичной цифры), а также процедура вывода символа Write_char.



CX - счетчик десятичных цифр в числе

Рис. 11. Алгоритм вывода десятичного числа

Процедура Test_write_dec тестирует процедуру Write_dec с помощью числа 12345 (которое транслятор переводит в слово 3039h):

```
Test_write_dec   PROC    NEAR
                 MOV     DX, 12345
                 CALL    Write_dec
```

```
                INT     20h             ; Возврат в DOS
Test_write_dec  ENDP
```

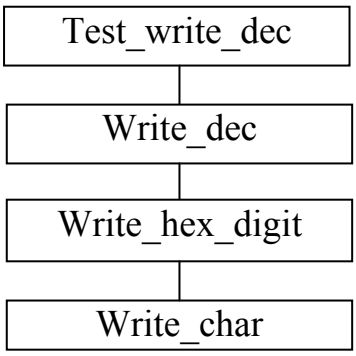


Рис. 12. Дерево подпрограмм для программы вывода десятичного числа

Запись на ассемблере

При кодировании процедуры Write_dec можно использовать два следующих приема программирования. (Хотя выгода от их применения небольшая, они широко используются на практике.)

Целью первого приема является обнуление ячейки (слова или байта). Для этого используется логическая инструкция XOR – "исключающее ИЛИ". Например, инструкция XOR AX,AX обнуляет регистр AX. Она сравнивает оба операнда побитно, и если один из битов равен 1, то и в соответствующий бит результата записывается 1. Если оба сравниваемых бита содержат 0, или оба содержат 1, то в результирующий бит записывается 0. Например:

```
        1011    0101
XOR     1011    0101
        0000    0000
```

Второй прием используется для того, чтобы проверить на равенство 0 слово или байт. Вместо инструкции CMP AX,0 можно записать инструкцию OR AX,AX , используя далее или инструкцию условного перехода JNE (от "Jump if Not Equal" – перейти, если не равно) или JNZ (перейти, если не нуль).

Логическая инструкция OR ("ИЛИ") побитно сравнивает оба операнда и записывает 1 в бит результата тогда, когда хотя бы в одном из двух сравниваемых битов есть 1. Данная инструкция устанавливает флаг нуля только тогда, когда все биты результата содержат 0.

Операция OR, выполненная по отношению к одному и тому же числу, дает в результате это же число:

```
        1011    0101
```

```
OR      1011        0101
--------------------------------
        1011        0101
```

Инструкция OR также полезна при установке одного бита в 1. Например, мы можем установить бит 3: .

```
        1011        0101
OR      0000        1000
---------------------------------
        1011        1101
```

В н е с и т е изменения в полученный ранее исходный файл Video_io.asm. Во-первых, с помощью текстового редактора удалите процедуру Test_write_hex и на ее место запишите тестовую процедуру Test_write_dec.

Во-вторых, получите текст на ассемблере процедуры Write_dec, используя для кодирования этапов ее алгоритма "CX ← 0" и "Частное = 0" описанные выше два приема. Процедуру Write_dec поместите в файл Video_io.asm после процедуры Test_write_dec.

В-третьих, в конце файла Video_io.asm измените оператор END, чтобы читалось: END Test_write_dec , т.к. теперь главной процедурой является Test_write_dec.

Внеся изменения, ***п р о д е л а й т е*** с Video_io все необходимые шаги, чтобы получить .com-файл. После этого протестируйте программу. В случае ошибки проверьте исходный файл. Если это не поможет, то для поиска ошибки используйте Debug.

Выполняя тестирование, будьте осторожны при проверке граничных условий. Первое граничное условие 0 не представляет трудности. Другое граничное условие – 65535 (FFFFh), которое вам лучше проверять с помощью Debug. ***З а г р у з и т е*** Video_io.com в Debug, напечатав "Debug Video_io.com", и замените 12345 (3039h) по адресу 101h и 102h на 65535 (FFFFh). Напомним, что Write_dec работает с беззнаковыми числами.

Раздельное ассемблирование

Для выполнения тестирования процедуры Write_dec нам пришлось одну тестовую процедуру в файле Video_io.asm заменить на другую, которая после завершения тестирования также больше не нужна. Очень неудобно вносить изменения в файл с готовыми (или почти готовыми процедурами) с целью добавления в него процедур временного назначения. Выход заключается в размещении тестирующей процедуры в отдельном исходном файле. Когда данная процедура становится ненужной, соответствующий файл уничтожается. Другой причиной размещения исходной программы в нескольких файлах является большой объем программы.

Далее будем понимать под файловой структурой исходной программы

перечень файлов, содержащих текст программы на языке программирования. А также перечень процедур, входящих в состав каждого файла. Кроме процедур файлы могут содержать переменные - области ОП для размещения данных, которым в исходной программе присвоены символьные имена. Примеры применения переменных встретятся нам в дальнейшем.

Следует отметить, что в отличие от дерева подпрограмм файловая структура не влияет на машинную программу. Данная структура создается для исходной программы с целью обеспечения удобства в программировании. Нескольким программам, имеющим разные файловые структуры, может соответствовать одна и та же результирующая машинная программа.

При выполнении данной работы мы добавили в файл Video_io.asm короткую тестовую процедуру Test_write_dec. Теперь уберем ее оттуда и поместим в собственный отдельный файл Test.asm. В результате получим файловую структуру программы, приведенную на рис.13.

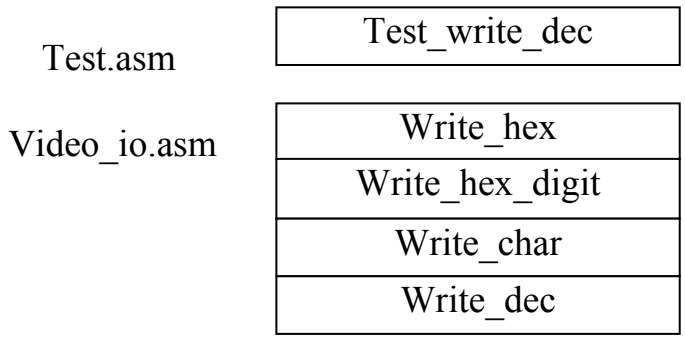


Рис. 13. Файловая структура исходной программы

Несмотря на то, что файловая структура исходной программы не влияет на результирующую машинную программу, она влияет на процесс получения этой программы. Это обусловлено тем, что каждый исходный файл преобразуется транслятором-ассемблером в объектный файл с расширением .obj совершенно независимо от других исходных файлов. Для того, чтобы в дальнейшем редактор связей мог связать эти объектные модули в единый загрузочный модуль типа .com (или типа .exe), транслятор должен передать ему некоторую служебную информацию. Исходные данные для этой информации сообщаются транслятору программистом с помощью псевдооператоров ассемблера.

Рассмотрим эти псевдооператоры на примере файла Test.asm:

```
Code_seg    SEGMENT        PUBLIC
            ASSUME    CS:Code_seg
            ORG       l00h
```

```
                EXTRN       Write_dec:NEAR
Test_write_dec  PROC    NEAR
                MOV     DX, 12345
                CALL    Write_dec
                INT     20h                     ;  Возврат в DOS
Test_write_dec  ENDP
Code_seg    ENDS
        END     Test_write_dec
```

Обратите внимание, что после SEGMENT появилось слово PUBLIC, которое сообщает транслятору о том, что мы хотим, чтобы этот сегмент (Code_seg) был скомбинирован с одним из сегментов, имеющим такое же имя. Транслятор-ассемблер передает эту информацию редактору связей (Tlink), который соединяет разные файлы в один.

Наш файл содержит также псевдооператор EXTRN. Выражение

```
EXTRN   Write_dec:NEAR
```

сообщает транслятору:

1) процедура Write_dec находится в другом исходном файле;
2) процедура Write_dec объявлена в своем файле как NEAR. Таким образом ассемблер генерирует для этой процедуры "близкий" вызов ("NEAR CALL"). "Дальний" вызов (“FAR CALL”) был бы сгенерирован в том случае, если бы мы поместили после Write_dec слово FAR.

З а п и ш и т е файл Test.asm на диск и внесите в файл Video_io.asm следующие изменения:

1) удалите из Video_io.asm процедуру Test_write_dec, т.к. мы поместили ее в файл Test.asm ;
2) удалите из Video_io.asm выражение ORG l00h, т.к. мы перенесли его в файл Test.asm, который теперь содержит главную процедуру программы;
3) поместите слово PUBLIC после SEGMENT:

   ```
   Code_seg   SEGMENT   PUBLIC
   ```

   для того, чтобы компоновщик знал о том, что он должен скомбинировать этот сегмент с таким же сегментом в Test.asm ;
4) перед каждой процедурой, расположенной в данном файле, но которая может вызываться из других файлов, записывается псевдооператор PUBLIC , например:

   ```
   PUBLIC      Write_hex_digit
   ```
5) измените " END Test_write_dec " в конце Video_io.asm на просто "END", т.к. мы переместили главную процедуру в Test.asm.

После внесенных изменений исходный файл Video_io.asm выглядит так:

```
Code_seg        SEGMENT   PUBLIC
                ASSUME    CS:Code_seg
```

```
                PUBLIC      Write_dec
                  .   .   .
Write_dec                   ENDP
                PUBLIC      Write_hex
                  .   .   .
Write_hex                   ENDP
                PUBLIC      Write_hex_digit
                  .   .   .
Write_hex_digit                 ENDP
                PUBLIC      Write_char
                  .   .   .
Write_char          ENDP
Code_seg    ENDS
                    END
```

А с с е м б л и р у й т е эти два файла так же, как вы раньше ассемблировали Video_io.asm. В результате Вы получите файлы Test.obj и Video_io.obj. Используйте следующую команду, чтобы произвести связывание этих программ в один файл (загрузочный модуль):

C > TLINK Test Video_io /t

TLINK использует имя первого файла в качестве названия результирующего .com-файла, так что теперь у нас есть файл Test.com. Результат аналогичен .com-файлу, полученному раньше из одного файла Video_io.asm, когда он содержал главную процедуру Test_write_dec .

Задание

Требуется написать на ассемблере и отладить программу, выполняющую те же функции, что и программа в задании к работе 2, с тем лишь отличием, что на экран выводятся содержимые регистров BP и DI не только в двоичной и шестнадцатеричной, но и в десятичной системе счисления.

Примечание 1. Файловая структура программы должна включать два файла типа .asm. В одном из них содержатся главная подпрограмма и тексты выводимых сообщений. Все остальные процедуры содержатся во втором файле.

Примечание 2. Все процедуры должны иметь вводные и текущие комментарии.

Запишите полученные .asm- и .com-файлы в каталог LAB3 на дискету для пересылки.

Лабораторная работа N 4

ДАМПИРОВАНИЕ ПАМЯТИ

Цель работы

В процессе выполнения работы решается задача разработки программы на ассемблере, выполняющей дампирование (вывод на экран) содержимого памяти подобно тому, как это делает Debug при выполнении команды D. Целью выполнения работы является изучение новых приемов программирования на ассемблере, в том числе, изучение новых исполнительных операторов и псевдооператоров. Кроме того, преследуется цель развития навыков проектирования программ. При этом рассматривается универсальный подход к разработке интерактивных программ, управляемых с помощью управляющих клавиш.

В отличие от предыдущих лабораторных работ, данная работа не является обязательной и результаты ее выполнения не пересылаются в ТМЦДО.

Дампирование шестнадцати байтов

Разработку программы, выполняющей дампирование, целесообразно начать с создания процедуры, выполняющей вывод на экран содержимого шестнадцати байтов. Именно столько информации удобно разместить на одной строке экрана. Назовем данную процедуру Disp_line . Для каждого из 16-ти байтов она выводит на экран соответствующее шестнадцатеричное представление, разделяя два соседних числа пробелами.

Ниже приведен файл Disp_sec.asm, который содержит начальную версию процедуры Disp_line:

```
Cgroup      GROUP   Code_Seg, Data_seg      ; Объединяет два сегмента
              ASSUME          CS:Cgroup, DS:Cgroup
Code_seg    SEGMENT   PUBLIC
                    ORG           l00h
              EXTRN       Write_hex:NEAR
              EXTRN       Write_char:NEAR
;
;         Процедура дампирует 16 байт памяти в одну строку
;                          шестнадцатеричных чисел
;         ---------------------------------------------------------------------
; Вызовы:    Write_hex,   Write_char
; Чтение:     Sector
;
 Disp_line    PROC      NEAR
```

```
                XOR     BX, BX              ; Обнуление  BX
                MOV     CX, 16              ; Счетчик  байтов
Hex_loop:       MOV     DL, Sector[BX]      ; Получить один байт
                CALL    Write_hex           ; Вывод  шестн. числа
                MOV     DL, ' '             ; Вывод  на  экран
                CALL    Write_char          ;   пробела
                INC     BX                  ; Возврат за следующим
                LOOP    Hex_loop            ;   байтом
                INT     20h                 ; Возврат в DOS
Disp_line   ENDP
Code_seg    ENDS
      ; ---------------------------------------------------------------------------
Data_seg    SEGMENT     PUBLIC
              PUBLIC    Sector
 Sector         DB    10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h   ; Образец
                DB    18h, 19h, 1Ah, 1Bh,1Ch,1Dh,1Eh,1Fh  ;    текста
Data_seg    ENDS
                END   Disp_line
```

Посмотрим внимательно на структуру программы. Она состоит из двух сегментов, расположенных друг за другом. Первым записан сегмент кодов, имеющий имя Code_seg, а за ним находится сегмент данных Data_seg. Выделение сегмента данных может показаться странным, т.к. при загрузке в память загрузочному модулю типа .com выделяется единственный сегмент памяти длиной 64K. Об этом говорит и псевдооператор

ASSUME CS:Cgroup, DS:Cgroup ,

который информирует транслятор о том, что и в регистр сегмента кодов CS и в регистр сегмента данных DS перед выполнением программы ОС загрузит начальный адрес (номер параграфа) общего сегмента с именем Cgroup.

Действительно, при создании программ типа .com сегмент данных можно не создавать вовсе. Так часто и делают, т.е. размещают данные в сегменте кодов. При этом в начале сегмента записывают оператор безусловного перехода, осуществляющий переход через область данных, расположенную после оператора перехода, на начало области исполнительных операторов программы.

Использование сегмента данных фактически означает, что мы представляем единый программный сегмент в виде двух непересекающихся частей. Это особенно полезно при раздельной трансляции программы. Например, допустим, что исходная программа состоит из двух исходных модулей. Каждый такой модуль находится в своем файле и транслируется отдельно от другого. Если данные в каждом из файлов выделить в свой сегмент данных, то в результате выполнения редактора связей обе области

данных будут объединены в единую область, отдельную от области кодов. При отсутствии сегментов данных подобного объединения данных сделать нельзя. Поэтому далее вначале любого исходного файла будем записывать два псевдооператора:

```
Cgroup      GROUP     Code_seg, Data_seg
            ASSUME    CS:Cgroup, DS:Cgroup
```

Псевдооператор GROUP группирует различные сегменты в один, названный именем, которое мы указали перед GROUP. Следовательно, записанный нами GROUP соединяет сегменты Code_seg и Data_seg в один 64-килобайтный сегмент, имеющий имя Cgroup.

Обратим внимание на использование слова PUBLIC. Это использование различно для сегментов и для других программных объектов – процедур и областей данных (переменных). Применительно к сегменту слово PUBLIC означает, что при связывании файлов редактором связей данный сегмент будет объединен с сегментами в других файлах, имеющими то же имя и тип PUBLIC. Порядок следования объединяемых сегментов определяется порядком следования объектных модулей (файлов) в команде вызова редактора связей TLINK .

Применительно к процедуре или к переменной слово PUBLIC означает, что процедура (переменная) определена в данном файле, а может быть использована не только в нем, но и в любом другом, в котором она указана с помощью псевдооператора EXTRN. В нашей программе тип PUBLIC имеет переменная Sector, а тип EXTRN – процедуры Write_hex и Write_char.

Что касается исполнительных операторов программы, то новым является использование относительной регистровой адресации в операторе

```
Hex_loop:    MOV   DL, Sector[BX]
```

Допустим, что байт, отмеченный меткой Sector, имеет смещение 431 относительно самого первого байта сегмента Cgroup (431 = 100h + длина сегмента кодов). Тогда при выполнении на ЦП машинной инструкции, соответствующей записанному оператору, реальный адрес второго операнда будет определен по формуле:

R = (DS) x 16 + 431 + (BX)

Задав (BX) = 0 мы получим адрес байта с меткой Sector, в который транслятором было помещено число l0h. Меняя содержимое регистра BX от 0 до 15 мы можем с помощью приведенного выше оператора MOV записать в регистр DL содержимое любого из 16-ти байтов, проинициализированных по нашей просьбе транслятором.

З а п и ш и т е файл Disp_sec.asm и внесите следующие изменения в полученный ранее файл Video_io.asm. Удалите псевдооператор ASSUME и поместите в начало этого файла две строки:

```
Cgroup      GROUP     Code_seg
            ASSUME    CS:Cgroup
```

Данные строки получены путем упрощения тех двух строк, которые мы договорились записывать в начале любого файла, с учетом того, что сегмент Data_seg в файле Video_io.asm не нужен. Фактически данные операторы означают, что метки Cgroup и Code_seg идентичны.

В ы п о л н и т е ассемблирование файлов Disp_sec.asm и Video_io.asm, а затем применив TLINK создайте файл Disp_sec.com . Если после запуска программы вы не увидите:

10 11 12 13 14 15 16 17 18 19 IA 1B 1C 1D 1E 1F ,

то вернитесь назад и найдите ошибку.

Дампирование 256 байтов памяти

После того, как мы сделали первую версию процедуры вывода на экран шестнадцатеричного представления 16-и байтов памяти, перейдем к изготовлению программы, выполняющей дамп 256 байтов памяти.

Число 256 обусловлено следующим. Во-первых, задачей лабораторной работы является создание программы, позволяющей выполнять дампирование текстовой информации, находящейся в сегменте памяти. Во-вторых, объем сегмента равен 65536 байт и 256 есть наибольшее число, которое одновременно отвечает двум требованиям : 1) является делителем числа 65536 ; 2) содержимое 256 байтов одновременно умещается на экране. Далее будем называть такой объем памяти сектором. Поэтому назовем процедуру, выполняющую дамп 256 байтов, как Disp_sector .

До сих пор мы делали вывод символов, которые размещались на одной строке экрана. При выполнении дампа 256 байтов нам потребуются 16 строк. Поэтому нам потребуется выполнять переход с одной строки экрана на следующую строку. Для осуществления такого перехода достаточно "вывести" на экран два управляющих символа кода ASCII. Первый символ имеет код 0Dh и называется "возврат каретки". В результате его "вывода" последующий вывод на экран начнется с самой левой позиции строки. Второй управляющий символ имеет код 0Ah и называется "перевод строки". Его "вывод" и реализует перевод строки экрана.

Назовем процедуру, выполняющую перевод строки как Send_crlf (crlf означает "Carriage Return - Line Feed." - "Возврат каретки – Перевод строки").

Текст файла Cursor.asm , содержащего эту процедуру:

```
cr     EQU    13                    ; Возврат  каретки
lf     EQU    10                    ; Перевод  строки
Cgroup      GROUP      Code_seg
              ASSUME    CS:Cgroup
Code_seg    SEGMENT  PUBLIC
              PUBLIC     Send_srlf
;
;              Перевод  строки  экрана
```

```
;               -------------------------------
;
Send_crlf   PROC    NEAR
                PUSH      AX
                PUSH      DX
                 MOV      AH, 2              ; Функция  вывода
                 MOV      DL, cr             ; Выводимый  символ
                 INT      21h                ; Вывод  символа
                 MOV       DL, lf
                 INT      21h                ;         --- // ---
                 POP       DX
                 POP       AX
                  RET
Send_crlf     ENDP
Code_seg    ENDS
                END
```

Псевдооператор EQU указывает транслятору, что имя, расположенное слева от EQU, эквивалентно числу, стоящему от EQU справа. Поэтому везде в программе, где транслятор встретит данное имя, он его заменит на указанное число. Применение данного псевдооператора позволяет программисту записывать в программе вместо чисел более удобные их символьные обозначения. Например, вместо 13 записывать cr.

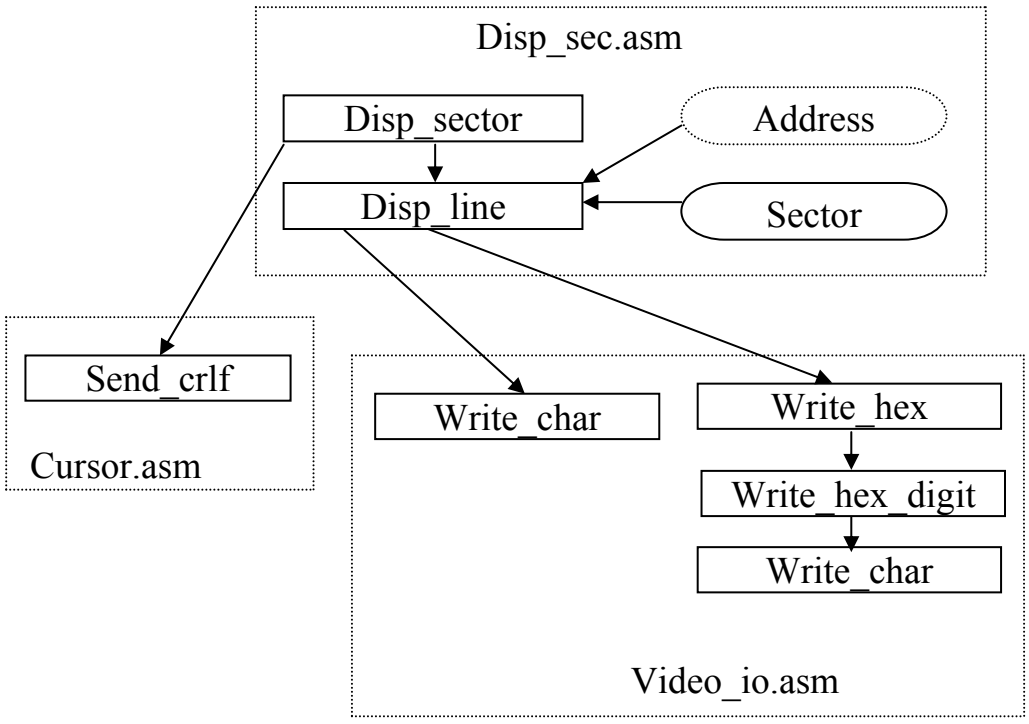


Рис. 14**.** Дерево подпрограмм и файловая структура программы вывода на экран сектора

На рис. 14 приведены дерево подпрограмм и файловая структура программы вывода на экран 256 байтов из переменной Sector. Переменная Address изображена пунктиром, т.к. в первой версии программы она не используется.

В в е д и т е файл Cursor.asm и скорректируйте файл Disp_sec.asm в соответствии с его новым вариантом:

```
Cgroup      GROUP   Code_seg, Data_seg
                    ASSUME  CS:Cgroup, DS:Cgroup
Code_seg    SEGMENT PUBLIC
                    ORG     l00h
                    EXTRN   Write_hex:NEAR
                    EXTRN   Write_char:NEAR
                    EXTRN   Send_crlf:NEAR
                    PUBLIC  Disp_sector
;
;           Отображает на экран сектор (256 байт)
;           ------------------------------------------------------
;  Вызовы:  Disp_line ,  Send_crlf
;
 Disp_sector  PROC        NEAR
                XOR       DX, DX        ; Начало Sector
                MOV       CX, 16        ; Число строк 16
   M1:          CALL      Disp_line     ; Вывод строки
                CALL      Send_crlf     ; Перевод строки
                ADD       DX, 16        ; Номер следующего байта
                LOOP      M1            ; Проверка числа строк
                INT       20h           ; Выход в DOS
 Disp_sector  ENDP
;
; Процедура дампирует 16 байт памяти в одну строку шестнадцат. чисел
; ------------------------------------------------------------------------------------------
;  Входы :     DX – номер первого байта строки в Sector
;  Вызовы :    Write_char,  Write_hex
;  Читается :  Sector
;
Disp_line   PROC      NEAR
            PUSH      BX
            PUSH      CX
            PUSH      DX
            MOV       BX,DX             ; В BX номер первого байта
            MOV       CX, 16            ; Счетчик байтов
```

```
Hex_loop:  MOV     DL, Sector[BX]     ; Получить один байт
           CALL    Write_hex          ; Вывод шестнад. числа
           MOV     DL,' '             ; Вывод на экран
           CALL    Write_char         ;  пробела
           INC     BX                 ; Возврат за
           LOOP    Hex_loop           ;  следующим байтом
           POP     DX
           POP     CX
           POP     BX
           RET
Disp_line  ENDP
Code_seg   ENDS
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Data_seg   SEGMENT  PUBLIC
                    PUBLIC   Sector
  Sector       DB   16  DUP(10h)      ; Первая строка байтов
               DB   16  DUP(11h)
               . . . . . .
               DB   16  DUP(1Fh)      ; Последняя строка байтов
Data_seg   ENDS
           END  Disp_sector
```

Обратите внимание на изменения, которые мы внесли в процедуру Disp_line. Теперь она имеет входной параметр, передаваемый в регистре DX. Это номер байта в поле Sector, начиная с которого следует вывести на экран 16 байтов. Есть и другие изменения в данной процедуре, не требующие пояснений.

В ы п о л н и т е ассемблирование и проведите компоновку (связывание) трех файлов: Disp_sec.asm, Video_io.asm и Cursor.asm. В этом списке Disp_sec.asm должен быть первым. Выполнив программу, вы должны получить изображение на экране:

```
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12
                    .  .  .
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F
```

Очистка экрана

Прежде, чем вывести на экран сектор, наша программа должна позаботиться об очистке экрана. Для выполнения этой операции мы обратимся за помощью к системной программе, называемой ROM BIOS.

Внутри каждого компьютера IBM PC имеются специальные микросхемы, называющиеся ROM ("Read Only Memory" - постоянное

запоминающее устройство, ПЗУ). В одной из этих микросхем записаны подпрограммы, обеспечивающие обмен информацией между ЦП и остальными частями компьютера. Т.к. подпрограммы, содержащиеся в ROM, обеспечивают ввод и вывод, то их часто называют BIOS ("Basic Input Output System" - базовая система ввода-вывода).

Операционная система, например DOS, использует подпрограммы BIOS для выполнения своих операций обмена с внешними устройствами. Разработчик прикладной программы может применять подпрограммы BIOS точно также, как и подпрограммы DOS, т.е. с помощью инструкций INT. Подпрограммы BIOS применяются в двух случаях: 1) соответствующая функция DOS отсутствует; 2) требуется повысить скорость выполнения функции.

Для очистки экрана будем использовать функцию BIOS – "Прокрутка экрана вверх". Она вызывается инструкцией INT l0h (функция 6). Данная функция требует задания следующих входных параметров:

(AL) – число строк, которые должны быть стерты внизу окна. Обычная прокрутка стирает одну строку. Нуль означает, что нужно стереть все окно;

(CH,CL} – строка и столбец левого верхнего угла окна;

(DH,DL) – строка и столбец правого нижнего угла окна;

(BH) – атрибуты (например, цвет), используемые для этих строк.

Таким образом, функция номер 6 десятого прерывания требует довольно много входной информации, даже если все сводится только к очищению экрана. Отметим, что она обладает мощными способностями: может очистить любую прямоугольную часть экрана – окно (“Window”). Текст процедуры Clear_screen, выполняющей очистку экрана:

```
                PUBLIC      Clear_screen
;
;                   Очистка экрана
;                   --------------------
;
Clear_screen    PROC        NEAR
                PUSH        AX
                PUSH        BX
                PUSH        CX
                PUSH        DX
                XOR         AL, AL          ; Очистить все окно
                XOR         CX, CX          ; Верхний левый угол в (0,0)
                MOV         DH, 24          ; Нижняя строка экрана -24
                MOV         DL, 79          ; Правая граница в 79 столбце
                MOV         BH, 7       ; Применить нормальные атрибуты
                MOV         AH, 6           ; Очистить
                INT         10h             ;    окно
```

```
                POP     DX
                POP     CX
                POP     BX
                POP     AX
                RET
Clear_screen    ENDP
```

З а п и ш и т е процедуру Clear_screen в файл Cursor.asm и протестируйте ее. Затем скорректируйте файл Disp_sec.asm, записав в начале процедуры Disp_sector инструкцию вызова процедуры очистки экрана. Не забудьте поместить в данный файл псевдооператор

```
EXTRN   Clear_screen: NEAR
```

и заново оттранслировать файл.

Запустив программу Disp_sec.com, можно убедиться, что экран очищается на весьма короткое время. Это обусловлено тем, что после завершения Disp_sec.com управление возвращается в DOS (или в Commander), которая восстанавливает на экране свою информацию. Для избежания этого в конце процедуры Disp_sector поместите инструкцию, выполняющую ввод символа с клавиатуры. Наличие такой инструкции позволит наблюдать на экране неискаженное изображение сектора до тех пор, пока не будет нажата любая клавиша. Впоследствии у нас отпадет потребность в подобном "торможении" программы.

С к о р р е к т и р у й т е программу (и соответствующие файлы) так, чтобы экранная строка дампа содержала бы сначала 4-х позиционный шестнадцатеричный адрес первого байта строки, далее один пробел, после которого шестнадцатеричное представление 16-и байтов памяти. Далее следует еще один пробел, после которого следует символьное представление этих же 16-и байтов (между символами пробелов нет). При этом требуется скорректировать процедуру Write_char так, чтобы вместо любого управляющего символа (код ASCII от 00h до 1Fh включительно) должен выводиться символ “.” . Пример экранной строки:

A17F 41 42 43 44 45 46 47 00 20 1F 80 81 82 83 84 85 ABCDEFG. .АБВГДЕ

Примечание 1. Переменная Address (слово) содержит начальный адрес 256-и байтовой области (рис. 14). Поэтому начальный адрес строки рассчитывается в процедуре Disp_line путем суммирования содержимого Address с содержимым регистра DX. (Далее будут созданы процедуры, которые загружают в переменную Sector редактируемый фрагмент ОП, а в переменную Address – начальный адрес (внутрисегментное смещение) этого фрагмента.) Не забудьте определить переменную Address в файле Disp_sec.asm, задав ей первоначальное значение.

Примечание 2. Перед оператором с меткой Hex_loop в Disp_line запишите оператор PUSH BX, который сохранит в стеке номер первого

байта выводимой строки. Этот номер пригодится при выводе на экран символов (для его получения не забудьте записать оператор POP BX).

Функции переписки сектора

Ранее мы использовали для хранения дампируемого сектора (256 байт) область Sector. В принципе можно было бы не использовать эту область, а напрямую работать с требуемым сектором памяти. Но в этом случае затрудняется преобразование в будущем нашей программы дампирования в текстовый редактор, так как любая ошибка в редактировании мгновенно отразилась бы на оригинале. Поэтому мы будем копировать текущий сектор памяти в область Sector, а после выполнения редактирования (или его имитации) осуществим обратное копирование.

На рис.15 приведено дерево подпрограмм для программы, выполняющей различные виды функций переписки сектора. Набор этих функций определяется потребностями пользователя разрабатываемой программы. Каждая из функций программно реализуется одной из следующих процедур:

1) Copy_sector – переписывает содержимое переменной Sector в область памяти, начальный адрес (сегментное смещение) которой находится в переменной Address ;
2) Init_sector помещает в переменную Sector начальный сектор сегмента нашей программы. Кроме того, эта процедура помещает 0 в переменную Address ;
3) Prev_sector помещает в Sector предыдущий сектор памяти, а в переменную Address записывает начальный адрес этого сектора ;
4) Next_sector помещает в переменные Sector и Address соответственно содержимое и начальный адрес следующего сектора памяти;
5) N_sector загружает в Sector N-й сектор сегмента программы. При этом значение N (0≤N≤255), умноженное на 256, записывается в переменную Address.

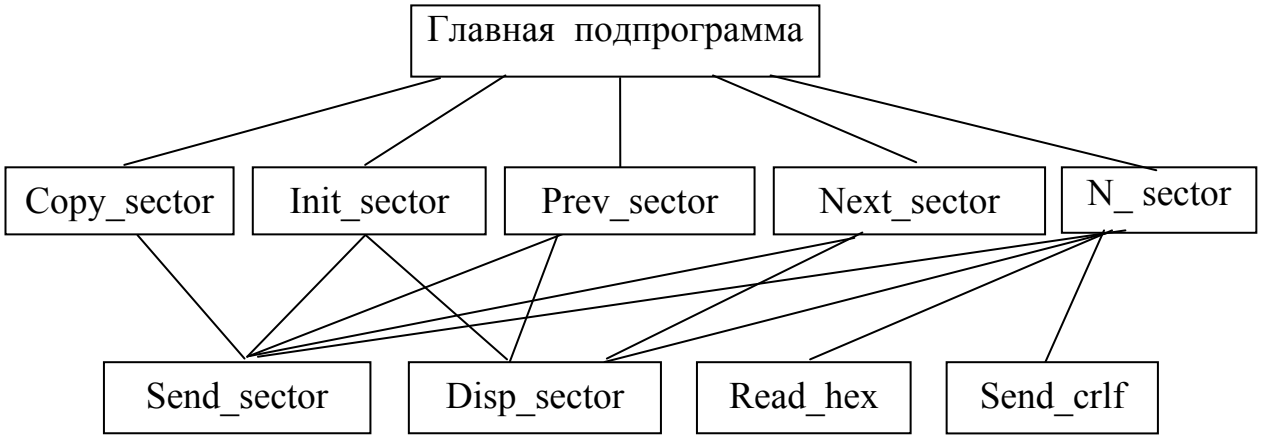


Рис. 15. Дерево подпрограмм для программы, выполняющей функции переписки сектора

Процедура Send_sector

Как видно из рис. 15, все процедуры, выполняющие функции по переписке сектора, пользуются услугами процедуры Send_sector, которая записывает содержимое заданного сектора памяти в качестве содержимого другого заданного сектора. Send_sector имеет два входных параметра: регистр SI содержит адрес (внутрисегментное смещение) сектора-источника, DI – адрес сектора-приемника.

При написании процедуры Send_sector мы могли бы ограничиться уже известными нам инструкциями, но мы привлечем для этого ряд новых инструкций:

```
PUBLIC    Send_sector
;
;                Переписывает  сектор (256 байт)
;                ---------------------------------------
;     Входы :  SI  – начальный адрес сектора-источника
;                DI – начальный адрес сектора-приемника
;
Send_sector       PROC  NEAR
                  PUSH    CX
                  PUSHF             ; Сохранить флаг направления  DF
                  CLD               ; Сбросить этот флаг
                  MOV     CX, 256   ; В счетчике число байт
              REP MOVSB             ; Пересылка цепочки байт
                  POPF              ; Восстановить флаг  DF
                  POP     CX
                  RET
Send_sector   ENDP
```

Строковая (цепочечная) инструкция MOVSB занимает центральное место в Send_sector. Взятая без префикса REP, она выполняет:

1) пересылку байта из ячейки памяти с адресом в регистре SI в ячейку памяти, адрес которой находится в регистре DI ;
2) изменяет на единицу адреса в регистрах SI и DI.

Направление изменения (увеличение или уменьшение) адресов в SI и DI определяется значением флага направления DF в регистре флагов. Если DF=0, то адреса увеличиваются, а если DF=1, то уменьшаются. Сброс флага DF выполняет специальная инструкция CLD, а установку – инструкция STD. Так как флаг направления может использоваться и в процедуре, вызывающей нашу процедуру, то целесообразно сохранить его перед изменением в стеке, а затем восстановить его оттуда. Для этого

используются инструкции PUSHF и POPF, выполняющие запись в стек и извлечение оттуда регистра флагов.

Префикс повторения команды REP многократно усиливает мощность строковой инструкции. При этом команда MOVSB выполняется столько раз, каково содержимое регистра CX. При каждом выполнении содержимое CX уменьшается на единицу. Как только это содержимое станет равным нулю, то выполняется следующая за MOVSB инструкция.

Перед вызовом процедуры Send_sector необходимо в вызывающей ее программе загрузить в регистры SI и DI начальные адреса секторов памяти. Для этого можно использовать инструкцию LEA ("Load Effective Address" – "Загрузить эффективный адрес"). Например, в результате инструкции
LEA SI,Sector адрес самого первого байта области Sector будет помещен в регистр SI.

З а п и ш и т е процедуру Send_sector в файл Disp_sec.asm .

Алгоритмы процедур

На рис. 16 приведён алгоритм процедуры Init_sector, на рис.17 алгоритм Prev_sector, а на рис. 18 алгоритм N_sector. Что касается алгоритма процедуры Next_sector, то он очень похож на алгоритм Prev_sector. Основное отличие состоит в том, что номер текущего сектора N сравнивается не с 0, а с числом FFh.

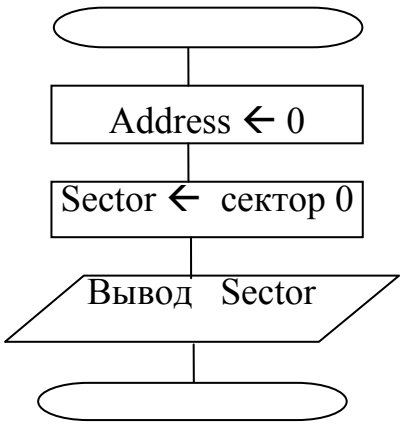
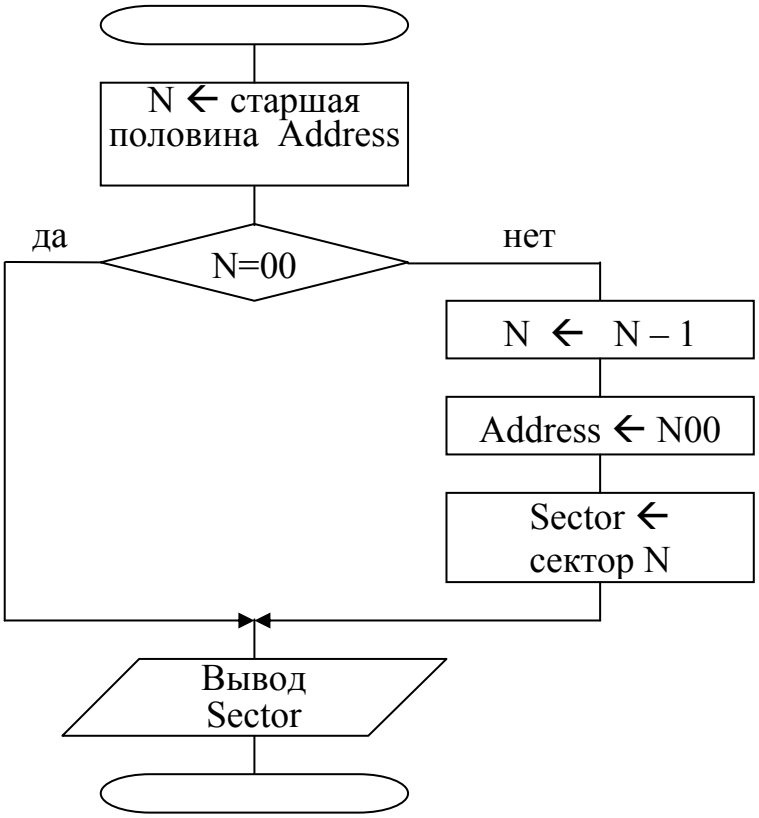


Рис. 16. Алгоритм процедуры Init_sector

З а п и ш и т е тексты процедур Copy_sector, Init_sector, Prev_sector, Next_sector, N_sector в файл Disp_sec.asm .

Примечание. Реализация этапа "Ввод N" в процедуре N_sector осуществляется путём вызова процедуры Read_hex, выполняющей ввод с клавиатуры 2-х значного шестнадцатеричного числа. Read_hex, в свою очередь, вызывает процедуру Read_hex_digit, выполняющую ввод шестнадцатеричной цифры (см. работу 2). ***З а п и ш и т е*** тексты обоих этих процедур в новый файл Kbd_io.asm .



N-номер текущего сектора

Рис. 17. Алгоритм процедуры Prev_sector

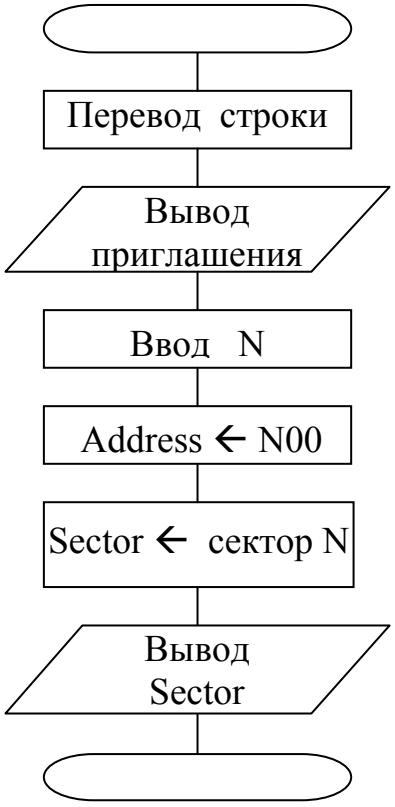
Применение для отладки программ .map-файлов

При выполнении программы TLINK создаётся не только файл загрузочного модуля программы, имеющий расширение .com (или .exe), но и файл с расширением .map, называемый картой загрузки. .map-файл записан в коде ASCII и поэтому его можно вывести на экран с помощью любого текстового редактора.

Если не использовать при вызове программы TLINK специального ключа /map, то получаемый в результате её выполнения .map-файл содержит весьма мало информации – начальные и конечные адреса расположения в памяти сегментов программы. ***У б е д и т е с ь*** в этом, найдя в каталоге файлов любой .map-файл и выведя его на экран.

Применение ключа /map позволяет получить подробную карту загрузки, содержащую начальные адреса расположения в памяти всех процедур и переменных программы, объявленных в исходной программе с помощью псевдооператора PUBLIC. Причём каждый из адресов встречается в тексте .map-файла дважды. В первом списке все процедуры и переменные

упорядочены по алфавитному порядку своих имен, а во втором – по порядку возрастания адресов. ***С о з д а й т е*** такую карту загрузки для любого ранее полученного .com-файла и проанализируйте её.



N - номер сектора (2-х значное шестнадцатеричное число)

Рис. 18. Алгоритм процедуры N_sector

Содержащаяся в .map-файле информация весьма полезна при отладке больших программ с помощью DEBUG. При этом адреса процедур можно использовать в качестве адресов останова при использовании команды G, а адреса переменных можно использовать для просмотра содержимого этих переменных с помощью команды D DEBUG. Следует отметить, что любые интересующие нас процедуры и переменные могут быть объявлены в программе с помощью псевдооператора PUBLIC .

В ы п о л н и т е отладку процедур, предназначенных для переписки сектора. Для этого в начале файла Disp_sec.asm временно поместите тестовую процедуру Imit:

```
Imit   PROC    NEAR
                CALL    Init_sector
                CALL    N_sector
                CALL    Prev_sector
                CALL    N_sector
                CALL    Next_sector
                INT       20h
Imit   ENDP
```

Вызовы процедуры N_sector обусловлены не только необходимостью отладки этой процедуры, но и позволяют во-первых, “затормозить” вывод секторов на экран, а во-вторых, позволяют сделать текущим любой интересующий нас сектор.

Примечание. При получении .com-файла получите карту загрузки и проведите её анализ.

Функции диспетчера команд

Почти любая полноценная программа является интерактивной, т.е. способной вести диалог со своим пользователем. Естественно, что наша программа дампирования памяти (которая в перспективе может развиться в редактор текстовой информации) также должна быть интерактивной. Она должна воспринимать команды пользователя, набираемые на клавиатуре и выводить на экран ответные сообщения. При этом принято называть сообщения пользователя (например, команды) входными сообщениями, а сообщения программы – выходными.

Распознавание команд пользователя поручим программному модулю (процедуре), называемому диспетчером команд. Данный модуль занимает центральное место в программной системе и координирует работу других модулей.

Ввод команды

Допустим, что наша программа выполняет всего шесть команд, каждой из которых соответствует своя управляющая клавиша:

<F1> – вывести на экран предыдущий сектор (256 байт) ОП ;

<F2> – записать скорректированный сектор в память ;

<F3> – вывести на экран следующий сектор ОП ;

<F4> – вывести на экран начальный сектор ОП ;

<F5> – вывести на экран сектор N, где 0<=N<=256 ;

<F10> – закончить работу программы.

Особенностью этих и многих других управляющих клавиш является то, что им соответствует не обычный, а расширенный код ASCII. В расширенном коде каждое из 256 кодовых значений может иметь наряду с

обычной интерпретацией второе смысловое значение. Например, код ASCII 3Bh обозначает символ “ ; ” , но этот же код соответствует и клавише <F1>. Аналогично код 44h соответствует и символу D и клавише <F10>. Таким образом, в расширенном коде ASCII предельное количество “символов” не 256, а 512.

Как разделить обычный и расширенный коды ASCII, чтобы не было путаницы? Допустим, что для ввода символа мы обращаемся к DOS с помощью инструкции INT 21h. В результате будет вызвана подпрограмма DOS, ожидающая нажатия клавиши. Если мы нажмем “обычную” клавишу, то данная подпрограмма передаст нашей программе в регистре AL соответствующий код ASCII. Если же мы нажмем управляющую клавишу, например, <F1>, то в AL будет возвращен 0. Если мы выполним инструкцию INT 21h повторно, то в AL будет возвращен код ASCII, соответствующий управляющему символу.

Ниже приведен текст процедуры Read_byte, выполняющей ввод с клавиатуры любого символа – обычного или расширенного. При этом код ASCII символа возвращается в регистре AL, а в регистре AH указывается тип символа (1 – обычный символ, -1 – символ расширенного кода). Обратите внимание, что вывод “эха” символа не производится.

```
                PUBLIC      Read_byte
;
;           Считывает  код  символа  с  клавиатуры
;           - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   Выход:  AL – код ASCII символа
;                AH – тип символа (1 – обычный символ;  -1 – символ
;                      расширенного кода)
;
Read_byte   PROC    NEAR
                    MOV     AH, 7           ;  Ввод символа
                    INT     21h             ;    без эха
                    OR       AL, AL         ;  Расширенный код ?
                    JZ       Extended       ;  Да
                    MOV     AH, 1           ; Обычный символ
                    JMP     Exit
Extended:           MOV     AH, 7           ; Ввод расширенного
                    INT     21h             ;   кода
                    MOV     AH, 0FFh        ;   AH ← -1
Exit:               RET
Read_byte   ENDP
```

П о м е с т и т е процедуру Read_byte в файл Kbd_io.asm, а затем выполните ее отладку, используя DEBUG. Для этого получите файл Kbd_io.com и наберите команду DOS:

DEBUG Kbd_io.com

Для запуска программы используйте команду DEBUG: G i , где i – адрес в листинге инструкции RET. (В файле Kbd_io.asm последний псевдооператор END должен временно содержать имя Read_byte.) Нажимая различные клавиши, проверьте правильность заполнения процедурой регистра AX.

Алгоритм диспетчера

На рис. 19 приведена блок-схема процедуры Dispatcher, выполняющей совместно с рассматриваемой далее процедурой Command функции диспетчера команд. Для того, чтобы обеспечить структурность алгоритма, мы, как и в одной из предыдущих работ, используем флаг переноса CF и операции над ним.

Кодирование процедуры Dispatcher не представляет особого труда. Три этапа ее алгоритма реализуются путем вызова ранее разработанных процедур. Этап «Вывод приглашения» реализуется путем вывода на экран строки символов или, даже, всего одного-двух символов, однозначно указывающих на то, что программа ожидает команд пользователя. Два этапа – «Редактирование символа» и «Выполнение команды» реализуются путем вызова соответственно процедур Edit_byte и Command, которых у нас пока нет.

Процедура Edit_byte не имеет выходных параметров и имеет единственный входной параметр: регистр DL содержит новый код ASCII редактируемого символа. Разработка данной процедуры выходит за рамки данной работы (она потребуется при преобразовании программы дампирования в редактор текстовой информации) и поэтому мы вынуждены заменить ее «заглушкой». Процедура–заглушка имеет то же имя и те же параметры, что и настоящая процедура. Но в отличие от нее «заглушка» не имеет или почти не имеет внутренних операторов.

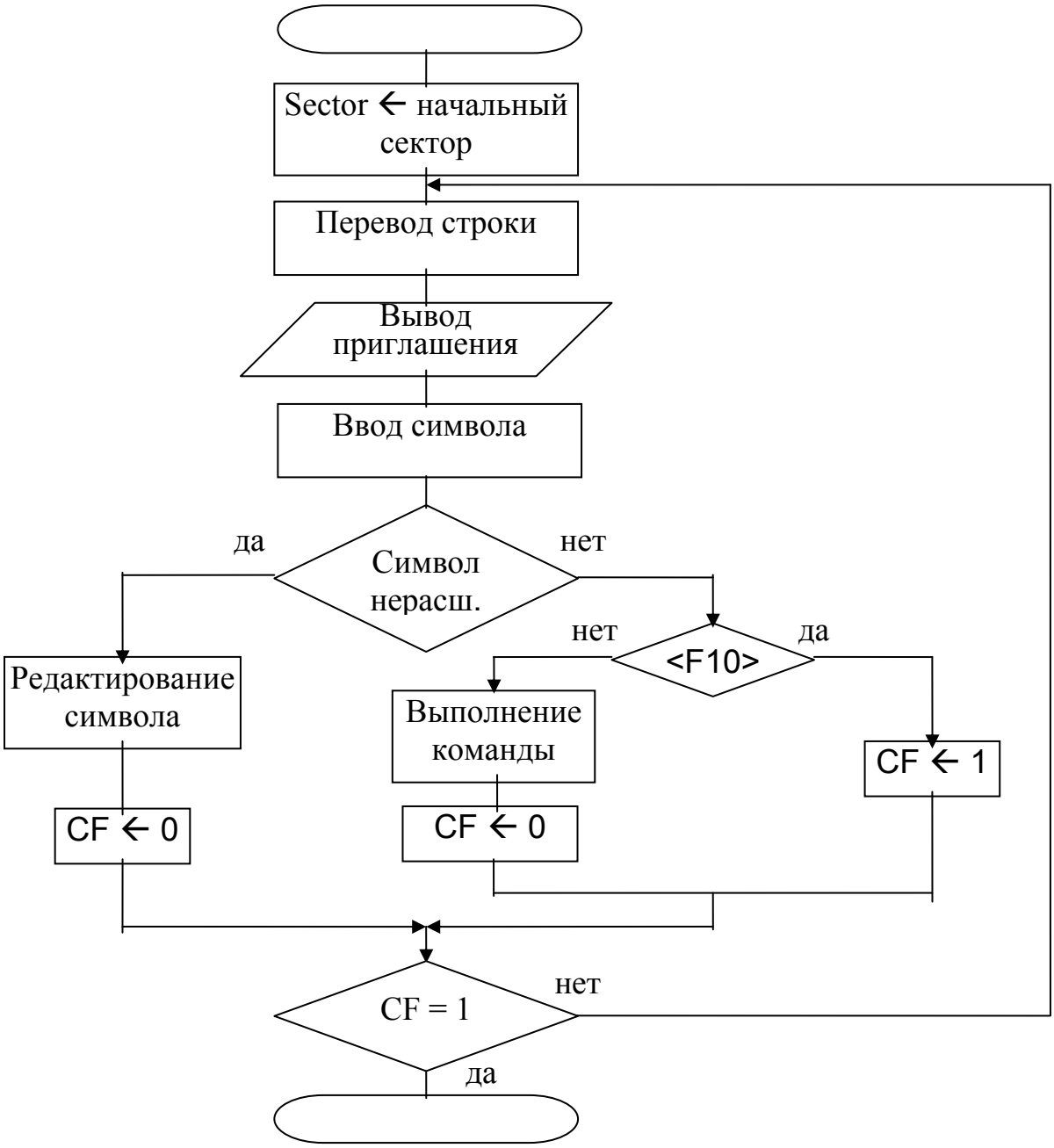
З а п и ш и т е процедуру-заглушку Edit_byte в файл Disp_sec.asm.

Выполнение команды

Этап “Выполнение команды” реализуется путем вызова процедуры Command. Данная процедура получает в регистре DL код ASCII, соответствующий команде, и в зависимости от этого кода вызывает процедуру, выполняющую заданное командой действие.

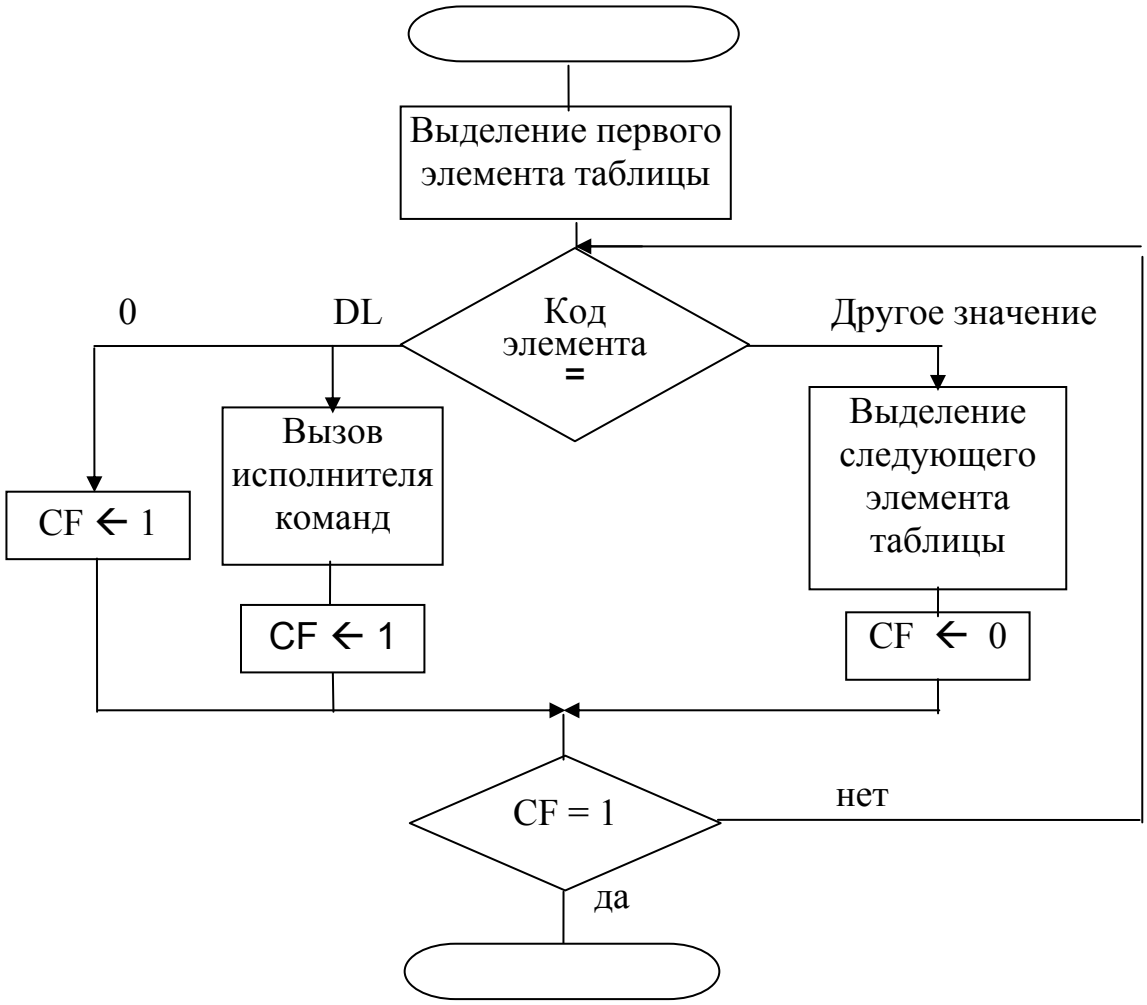
На рис. 20 приведена блок-схема процедуры Command. Центральной особенностью ее алгоритма является использование таблицы переходов. В данной таблице для каждой разрешенной команды пользователя отводятся

три байта. В первом байте находится код ASCII, соответствующий команде, а в двух других байтах – начальный адрес (внутрисегментное смещение) соответствующей процедуры. В последнем байте таблицы находится 0.



Sector – буфер для редактирования сектора
CF – признак завершения (0 – продолжить, 1 – окончить работу)

Рис. 19. Алгоритм процедуры Dispatcher



DL – код команды
CF – признак завершения (0 – продолжить, 1 – завершить работу)

Рис. 20. Алгоритм процедуры Command

Процедура Command совместно с процедурой Dispatcher обеспечивают выполнение алгоритма диспетчера команд. Поэтому обе процедуры, а также таблицу переходов Table мы поместим в один и тот же файл Dispatch.asm. Фрагмент этого файла, содержащий модули Command и Table :

```
Cgroup   GROUP   Code_seg,  Data_seg
                 ASSUME   CS: Cgroup,   DS: Cgroup
Code_seg  SEGMENT  PUBLIC
                 EXTRN   Send_crlf: NEAR
                 EXTRN   Init_sector: NEAR
                 EXTRN   Next_sector: NEAR
                 EXTRN   Prev_sector: NEAR
```

```
                    EXTRN   Edit_byte: NEAR
                    EXTRN   Copy_sector: NEAR
                    EXTRN   N_sector: NEAR
                    EXTRN   Read_byte: NEAR
                    ORG  100h
;
;       Координирует выполнение модулей редактора
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
Dispatcher  PROC  NEAR
.   .   .   .   .   .   .   .
Dispatcher   ENDP
;
;         Интерпретатор команд
;         - - - - - - - - - - - - - - - -
;   Входы:   DL - код ASCII, соответствующий команде
;   Чтение:  Table – таблица переходов
;
Command    PROC      NEAR
           PUSH   BX
           LEA    BX, Table              ; BX←адрес таблицы
M1:        CMP    BYTE  PTR [BX], 0      ; Конец таблицы ?
           JE     Not_in_table           ; Да, кода нет в таблице
           CMP    DL, [BX]               ; Это вход в таблицу?
           JE     Dispatch               ; Да, выполнить команду
           ADD    BX, 3                  ; Нет, переход к
           CLC                           ;   следующему
           JMP    M2                     ;   элементу таблицы
Dispatch:  INC    BX                     ; Адрес процедуры
           CALL   WORD  PTR [BX]         ; Вызов процедуры
           STC                           ;  CF ← 1
           JMP    M2
Not_in_table:  STC                       ;  CF ← 1
M2:        JNC    M1      ; Повторение для нового элемента таблицы
           POP    BX
           RET
Command    ENDP
Code_seg   ENDS
Data_seg   SEGMENT PUBLIC
;   Таблица содержит разрешенные расширенные ASCII-коды и
;    адреса процедур,  выполняющих соответствующие команды
  Table    DB        3Bh                              ; <F1>
           DW        OFFSET  CGROUP: Prev_sector
```

```
                DB      3Ch                                     ; <F2>
                DW      OFFSET  CGROUP: Copy_sector
                DB      3Dh                                     ; <F3>
                DW      OFFSET  CGROUP: Next_sector
                DB      3Eh                                     ; <F4>
                DW      OFFSET  CGROUP: Init_sector
                DB      3Fh                                     ; <F5>
                DW      OFFSET   CGROUP: N_sector
                DB      0                               ; Конец таблицы
Data_seg    ENDS
                    END    Dispatcher
```

Рассмотрим особенности этих модулей, обусловленные применением новых псевдооператоров PTR и OFFSET. Псевдооператор PTR используется для уточнения типа данного, обрабатываемого инструкцией программы. Обратите внимание, что таблица Table содержит и байты и слова. Так как мы работаем с двумя типами данных, то мы должны указать транслятору-ассемблеру, какой тип данных используется при применении инструкций CMP или CALL. В случае, если инструкция будет написана следующим образом: CMP [BX], 0 , то ассемблер не поймет, что мы хотим сравнивать – слова или байты.

Но если мы запишем инструкцию в виде: CMP BYTE PTR[BX],0 , то ассемблеру станет ясно, что BX указывает на байт, и что мы хотим сравнивать байты. Аналогично, инструкция CMP WORD PTR[BX],0 будет сравнивать слова. С другой стороны, инструкция CMP AL, [BX] проблем не вызывает, т.к. AL – это байтовый регистр, и ассемблер понимает без разъяснений, что мы хотим сравнивать байты.

Кроме того, как Вы помните, инструкция CALL может быть либо типа NEAR, либо типа FAR. Для задания адреса “близкого” вызова (“NEAR CALL”) требуется одно слово, в то время как для “дальнего” вызова требуется два слова. Например, инструкция CALL WORD PTR[BX] посредством «WORD PTR» говорит транслятору, что [BX] указывает на одно слово, поэтому ассемблер будет генерировать близкий вызов и использовать то слово, на которое указывает [BX], как адрес, который мы записали в Table. (Для дальнего вызова, который использует адрес, состоящий из двух слов, мы должны были бы использовать инструкцию: CALL DWORD PTR[BX] , где DWORD означает “Double Word” – двойное слово.)

Для того, чтобы получить адреса процедур, вызываемых диспетчером и содержащихся в таблице, мы применим новый псевдооператор OFFSET. Строка

DW OFFSET CGROUP: Prev_sector

сообщает транслятору о необходимости применить смещение процедуры

Prev_sector. Это смещение подсчитывается относительно начала группы CGROUP, и именно поэтому необходимо поставить “ CGROUP: ” перед именем процедуры. Если бы мы не поместили там CGROUP, то ассемблер подсчитывал бы адрес Prev_sector относительно начала сегмента кода, а это не совсем то, что нужно. (В данном конкретном случае CGROUP не необходим, т.к. в нашей программе сегмент кода загружается первым. Тем не менее, для ясности мы будем везде писать “ CGROUP: ” .)

Применение для реализации диспетчера команд таблицы переходов значительно увеличивает его гибкость, т.е. пригодность к модификации. В самом деле, мы легко можем добавить в нашу программу новые команды, набираемые на клавиатуре. Для этого достаточно записать процедуру, выполняющую новую команду, в подходящий файл, и поместить новую точку входа в Table.

Задание

З а п и ш и т е файл Dispatch.asm на диск и выполните отладку программы. Отлаженная программа должна правильно реагировать на нажатие любой клавиши.

5. КОНТРОЛЬНАЯ РАБОТА N 1

ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ В ЭВМ И ЭЛЕМЕНТЫ ЯЗЫКА АССЕМБЛЕРА

Контрольная работа №1 компьютерная и выполняется в диалоге с контролирующей программой. Она включает 10 заданий. Ниже приводятся примеры решений заданий и методические рекомендации по их выполнению.

Задание 1. Получить двоичное представление заданного целого десятичного числа.

Указания. Для преобразования десятичного числа в двоичное можно использовать метод вычитаний [1, п.2.1].

Пример. Получить двоичное представление десятичного числа 5000.

Решение. Выполним вычитания:

```
_ 5000
  4096  (бит 12)
 _904
   512  (бит 9)
_ 392
```

256 (бит 8)
_ 136
128 (бит 7)
_ 8
8 (бит 3)
0

Заполним биты двоичного числа, начиная с младшего:

1001110001000 (ответ).

Задание 2. Найти разность двух заданных положительных двоичных чисел.

Указания. Вычитание двух положительных чисел следует выполнить сложением первого из них с дополнением второго (отрицательного) числа [1, п.2.1]. Прежде, чем получать дополнение отрицательного числа, необходимо исходное положительное число дополнить слева незначащими нулями до байта или слова (в зависимости от величины числа). Иначе – знаковый бит числа (бит 7 для байта, бит 15 для слова) будет отсутствовать.

Пример. Найти разность двоичных чисел 1010100100111 и 1100100100111001.

Решение. Расширим второе число до 16 бит: 0110010010011001 . Найдем дополнительный код этого же числа, но взятого со знаком “-“ :

1) инвертируем все биты: 1001101101100110
2) прибавляем 1: + 1
1001101101100111

Суммируем первое число с полученным дополнением:

1010100100111
+ 1001101101100111
1011000010001110 (ответ)

Задание 3. Найти разность двух заданных положительных шестнадцатеричных чисел. (Из большего числа вычесть меньшее.)

Указания. Вычитание шестнадцатеричных чисел выполняется аналогично вычитанию десятичных чисел с той лишь разницей, что занимаемая из старшего разряда единица, в текущем разряде составляет не 10, а 16 единиц [1, п. 2.2].

Пример. Найти разность шестнадцатеричных чисел A74F и 8E5C.

Решение: _A74F
8E5C
18F3 (ответ).

Задание 4. Пусть в данный момент времени некоторые регистры содержат:

(регистр 1) =XXXXh, . . . , (регистр n) = XXXXh

Каков (в шестнадцатеричной системе) физический адрес ячейки ОП, содержащей:

1) младший байт следующей исполняемой на ЦП инструкции;
2) младший байт вершины стека;
3) байт данных, обрабатываемый текущей инструкцией [инструкция].

Указания. Для получения физического адреса искомой ячейки необходимо просуммировать содержимое соответствующего регистра сегмента, умноженное на 16 (10h), с содержимым регистра, в котором находится внутрисегментное смещение [1, п. 3.3, 5.1].

Пример. Пусть в данный момент времени некоторые регистры содержат:

. . . . (DS) = 3FA5h . . . (BX) = 4A84h

Найти физический адрес ячейки ОП, содержащей байт данных, обрабатываемый текущей инструкцией: MOV [BX], AL

Решение.

R = (DS) x 16 + (BX) = 3FA5 x 10h + 4A84 = 3FA50 + 4A84 =

= **444D4** (ответ).

Задание 5. Пусть в данный момент времени некоторые регистры содержат:

(регистр 1) =XXXXh, . . . , (регистр n) = XXXXh

Каково будет содержимое указателя команды (или указателя стека) в результате выполнения следующих машинных инструкций: [инструкции с указанием их длины].

Указания. Каждая выполняемая инструкция обработки данных влияет на содержимое указателя команды IP в соответствии со своей длиной, а инструкция передачи управления – в соответствии со своим адресом перехода [1, п. 3.2].

На указатель стека SP влияют только некоторые инструкции, а именно: PUSH, POP, CALL, RET, INT, IRET, PUSHF, POPF. При этом следует учесть, что стек “растет” в сторону меньших адресов [1, п. 5.1].

Пример. Пусть в данный момент времени некоторые регистры содержат:

. . . (SP) = FE4A .

Каково будет содержимое указателя стека после выполнения следующих инструкций:

```
PUSH   AX     (длина 1 байт)
PUSH   BX     (длина 1 байт)
RET           (длина 1 байт)
CALL  200h    (длина 3 байта)
```

Решение. Инструкции PUSH и CALL добавляют в стек по одному слову (2 байта), а инструкция RET берет слово из стека. Следовательно, длина стека увеличится на 4 байта. Новое содержимое регистра SP:

(SP) = FE4A – 4 = **FE46** (ответ).

Задание 6. Записать содержимое (в шестнадцатеричной системе) заданного регистра, полученное в результате выполнения следующих операторов ассемблера [операторы].

Указания. Для подготовки к выполнению задания рекомендуется использовать информацию о логических операторах [1, п. 6.2.2].

Пример. Записать содержимое регистра BX, полученное в результате выполнения операторов:

```
MOV     BX, 0ABCDh
MOV     CX, 0707h
AND     BX, CX
```

Решение. Сначало запишем двоичное содержимое регистров BX и CX до выполнения операции AND:

(BX) = 1010101111001101
(CX) = 0000011100000111

Выполним операцию AND и переведем результат в шестнадцатеричную систему:

```
     1010101111001101
AND  0000011100000111
     0000001100000101
     [ 0 ][ 3 ] [ 0 ] [ 5 ]
```

Ответ: **305 .**

Задание 7. Указать номер неправильного оператора передачи данных в следующем списке операторов [операторы MOV].

Указание. Для подготовки к выполнению задания рекомендуется использовать информацию о операторах передачи данных [1, п. 6.2.3] и о вспомогательных псевдооператорах [1, п. 6.6].

Пример. Указать номер неправильного оператора в списке:

1) MOV AX, 0FFFFh
2) MOV AL, DL
3) MOV BH, OFFSET Table
4) MOV Arg, AX
5) MOV DL, BYTE PTR Table

Решение. Неправильным является оператор 3, так как для размещения смещения адреса (оно задается псевдооператором OFFSET) всегда требуются 16 бит памяти, то есть слово. А длина регистра BH – 8 битов.

Ответ: **3** .

Задание 8. Записать содержимое (в шестнадцатеричной системе) заданного регистра, полученное в результате выполнения следующих операторов ассемблера [операторы].

Указания. Для подготовки к выполнению задания рекомендуется использовать информацию о операторах сдвига [1, п. 6.2.5].

Пример. Записать содержимое регистра AL, полученное в результате выполнения операторов:

```
MOV     AL, 7Ah
STC
RCR     AL, 1
```

Решение. Определим содержимое регистра AL до выполнения оператора RCR:

(AL) = 7Ah = 01111010

Оператор RCR (циклический сдвиг вправо через перенос) выталкивает содержимое младшего бита первого операнда и помещает его в флаг CF, а прежнее содержимое CF заносит в старший бит операнда. Так как оператор STC поместил в CF 1, то в результате выполнения RCR:

(AL) = 10111101 = BDh , ответ: **BD** .

Задание 9. Записать содержимое (в десятичной системе) заданного регистра, полученное в результате выполнения следующих операторов ассемблера [операторы].

Указания. Для подготовки к выполнению задания рекомендуется использовать информацию о операторах условных переходов [1, п. 6.5.1] и о операторах цикла [1, п.6.5.3].

Пример. Записать содержимое регистра AX, полученное в результате выполнения операторов:

```
        XOR     AX, AX
        MOV     BX, 20
A1:     CMP     BX, 10h
        JBE     Next
        INC     AX
        DEC     BX
        JMP     A1
Next:  . . . . . .
```

Решение. В записанном фрагменте программы первые два оператора выполняют инициализацию цикла ПОКА, а последующие операторы кодируют сам этот цикл. Условием повторения цикла является условие (BX)>10h. Оператор условного перехода JBE (перейти, если меньше или равно) реализует выход из цикла. Этот оператор используется для переходов после сравнения беззнаковых чисел, каковыми являются операнды оператора CMP – (BX) и 10h.

При первом выполнении оператора CMP значение (BX)=20, что явно больше, чем 16 (10h). В результате первого выполнения цикла (AX)=1, а (BX)=19. В результате четвертого выполнения – (AX)=4, (BX)=16. Это выполнение цикла является последним, так как следующее выполнение оператора JBE реализует выход из цикла на оператор с меткой Next.

Ответ: **4**.

Задание 10. Предлагается определить длину зарезервированного участка памяти или длину выводимого на экран сообщения.

Указания. Для подготовки к выполнению задания рекомендуется использовать информацию о псевдооператорах определения данных [1, п. 6.4].

Пример. Указать длину (в байтах) участка памяти, резервируемого следующими операторами:

```
Mas1    DB   10  DUP (?)
Pere    DW  10, 20h, ‘Ab’
Text    ‘Hello’
```

Решение. ОП резервируется тремя псевдооператорами. Первый оператор резервирует 10 байтов, второй – 6 байтов (3 слова), третий – 5 байтов. Общее число резервируемых байтов памяти – **21** (ответ).

6. КОНТРОЛЬНАЯ РАБОТА N 2

РАЗРАБОТКА ПРОГРАММЫ НА АССЕМБЛЕРЕ

Введение

Целью выполнения данной работы является комплексная проверка навыков программирования на языке ассемблера.

Результаты работы представляются в виде совокупности следующих документов:

1) дерево подпрограмм;
2) файловая структура программы;
3) блок-схемы алгоритмов процедур;
4) исходный файл (файлы) программы;
5) загрузочный модуль программы.

Примеры деревьев подпрограмм приведены на рис. 10, 14, 15.
Примеры файловой структуры программы приведены на рис. 13 и 14.

Основным требованием к блок-схемам алгоритмов процедур является выполнение требований структурного программирования [1, п.7.3]. Примеры алгоритмов процедур приведены на рис. 5, 11, 19.

Основным требованием к исходным модулям (файлам) программы является наличие комментариев [1, п. 8.6]. Примеры исходных модулей приведены в описаниях лабораторных работ 3 и 4.

Дерево подпрограмм, файловая структура программы и блок-схемы процедур представляются на листах писчей бумаги, выполняются аккуратно хорошо понятным почерком. Допускается представление этих документов в виде файлов на дискете, полученных с помощью текстового редактора Word. Остальные документы представляются в виде файлов с расширениями .asm и .com на дискете.

Варианты контрольной работы №2

Вариант 1. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>.

Программа выводит на экран сумму этих чисел, представленную в десятичной и шестнадцатеричной системах счисления.

Вариант 2. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число N. По следующему запросу он вводит с клавиатуры N целых трехзначных положительных десятичных чисел, разделенных пробелами.

Программа выводит на экран сумму этих чисел, представленную в десятичной и двоичной системах счисления.

Вариант 3. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>.

Программа выводит наибольшее число из введенных, представленное в десятичной и двоичной системах счисления.

Вариант 4. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число N. По следующему запросу он вводит с клавиатуры N целых трехзначных положительных десятичных чисел, разделенных пробелами.

Программа выводит наибольшее число из введенных, представленное в десятичной и шестнадцатеричной системах счисления.

Вариант 5. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>.

Программа выводит наименьшее число из введенных, представленное в десятичной и шестнадцатеричной системах счисления.

Вариант 6. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число N. По следующему запросу он вводит с клавиатуры N целых трехзначных положительных десятичных чисел, разделенных пробелами.

Программа выводит наименьшее число из введенных, представленное в десятичной и двоичной системах счисления.

Вариант 7. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>.

Программа выводит последовательность этих же чисел, но записанных в обратном порядке и в шестнадцатеричной системе счисления.

Вариант 8. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число N. По следующему запросу он вводит с клавиатуры N целых трехзначных положительных десятичных чисел, разделенных пробелами.

Программа выводит последовательность этих же чисел, но записанных в обратном порядке и в двоичной системе счисления.

Вариант 9. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>.

Программа выводит эти же числа на экран в порядке возрастания величины числа, причем в шестнадцатеричной системе счисления.

Вариант 10. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число N. По следующему запросу он вводит с клавиатуры N целых трехзначных положительных десятичных чисел, разделенных пробелами.

Программа выводит эти же числа на экран в порядке возрастания величины числа, причем в двоичной системе счисления.

Вариант 11. По запросу программы пользователь вводит с клавиатуры последовательность целых трехзначных положительных десятичных чисел, разделенных пробелами. Ввод последовательности заканчивается нажатием <Enter>.

Программа выводит эти же числа на экран в порядке убывания величины числа, причем в двоичной системе счисления.

Вариант 12. По запросу программы пользователь вводит с клавиатуры целое положительное десятичное число N. По следующему запросу он вводит с клавиатуры N целых трехзначных положительных десятичных чисел, разделенных пробелами.

Программа выводит эти же числа на экран в порядке убывания величины числа, причем в шестнадцатеричной системе счисления.

Вариант 13. По запросу программы пользователь вводит с клавиатуры сообщение на русском языке, заканчивающееся символом “.” или “!” .

Программа выводит на экран это же сообщение, записанное только заглавными буквами.

Вариант 14. По запросу программы пользователь вводит с клавиатуры сообщение на русском языке, заканчивающееся символом “.” или “?” .

Программа выводит на экран это же сообщение, записанное только строчными (малыми) буквами.

Вариант 15. По запросу программы пользователь вводит с клавиатуры сообщение на английском языке, заканчивающееся символом “.” или “?” .

Программа выводит на экран это же сообщение, записанное только заглавными буквами.

Вариант 16. По запросу программы пользователь вводит с клавиатуры сообщение на английском языке, заканчивающееся символом “.” или “!” .

Программа выводит на экран это же сообщение, записанное только строчными (малыми) буквами.

Вариант 17. По запросу программы пользователь вводит с клавиатуры два целых четырехзначных положительных десятичных числа, разделенных знаком операции “+” или “-“ .

Программа выводит на экран результат операции в двух системах счисления – в десятичной и в двоичной (в дополнительном коде).

Вариант 18. По запросу программы пользователь вводит с клавиатуры два целых четырехзначных положительных десятичных числа, разделенных знаком операции “ * ” .

Программа выводит на экран результат операции умножения .

Вариант 19. По запросу программы пользователь вводит с клавиатуры два целых четырехзначных положительных десятичных числа, разделенных знаком операции “ / ” .

Программа выводит на экран результат операции деления (частное и остаток).

Вариант 20. По запросу программы пользователь вводит с клавиатуры два целых трехзначных положительных десятичных числа.

Программа выводит на экран сообщение о том, делится ли первое число на второе без остатка, а затем сообщение – делится ли без остатка второе число на первое.

Примечание

При вводе с клавиатуры десятичного числа следует учесть, что получение двоичного представления такого числа выполняется иначе по сравнению с шестнадцатеричным числом. При этом каждую очередную десятичную цифру следует умножить на вес позиции числа, а затем просуммировать результаты умножения. Например, при вводе 3-х значного числа первая цифра умножается на сто, вторая – на десять, а третья цифра берется без изменения.

ПРИЛОЖЕНИЕ 1
РАБОТА В СРЕДЕ MS-DOS

MS-DOS – дисковая операционная система фирмы Microsoft. Это простая однопользовательская, однопрограммная операционная система. К ее достоинствам относятся весьма малый объем занимаемой памяти, а также большое число разработанных для нее прикладных программ.

Запуск MS-DOS

Запуск MS-DOS производится или в самостоятельном режиме или в среде операционной системы Windows. Для запуска MS-DOS в самостоятельном режиме требуется, чтобы на логическом системном диске (c:) находились файлы, образующие MS-DOS:

1) Io.sys - содержит подпрограммы ввода-вывода;
2) Msdos.sys – содержит ядро операционной системы;
3) Command.com – интерпретатор команд MS-DOS.

Эти три файла никогда нельзя корректировать. Следующие два файла, относящиеся к MS-DOS, можно корректировать:

1) Config.sys – перечень сведений о устройствах, имеющихся в ЭВМ, и как с этими устройствами работать. Запуск драйверов устройств производится из этого файла;
3) Autoexec.bat – перечень команд ОС, которые она должна выполнить сразу же после своей загрузки в ОП.

Запуск MS-DOS в самостоятельном режиме обычно производится при отсутствии на системном диске операционной системы Windows. В этом случае запуск MS-DOS производится автоматически после включения питания. При наличии Windows с помощью дополнительных действий также можно запустить MS-DOS в самостоятельном режиме, но этого не делают по той причине, что Windows позволяет запускать MS-DOS более простыми способами:

1) в сеансе MS-DOS;
2) в режиме эмуляции MS-DOS.

В первом случае работа с MS-DOS производится в окне Windows и поэтому весьма удобна. Запуск этого режима производится:

< пуск > → < программы > → < сеанс MS-DOS >

В режиме эмуляции MS-DOS Windows очень искусно имитирует работу MS-DOS. Пользователь начинает работу с чистым экраном, никаких внешних признаков наличия Windows у него нет. Запуск этого режима производится:

< пуск > → < завершение работы > → < режим эмуляции MS-DOS >

Для выполнения программ, описываемых в данном пособии, явно предпочтительней режим “сеанс MS-DOS”.

Общие сведения о командах MS-DOS

После запуска MS-DOS любым из перечисленных выше способом на экране появляется ее приглашение, например, следующего типа:

C:\SIMP>

Это приглашение означает, что в текущий момент времени DOS ожидает от вас команды, и что она находится в точке файловой структуры – C:\SIMP ,

где SIMP – текущий каталог на логическом диске C: .

Строка экрана, в которой появилось приглашение, называется командной строкой. В этой строке вы набираете команду:

< имя программы > [< параметры >] ,

где часть команды в квадратных скобках не обязательна.

Имя программы – имя файла, содержащего загрузочный модуль программы (расширение имени файла - .com или .exe), или имя командного файла (расширение - .bat). Расширение имени файла в команде может отсутствовать.

Командный файл – файл, содержащий несколько команд MS-DOS. Он создается пользователем как обычный текстовый файл, имеющий любое собственно имя, но обязательно расширение имени - .bat . При этом каждая команда набирается на отдельной строке экрана. Один командный файл играет особую роль - Autoexec.bat. Он запускается автоматически (то есть без нашего участия) MS-DOS в самом начале ее работы. Обычно среди прочих команд этот файл содержит:

1) команду запуска подпрограммы PATH;
2) команды запуска драйверов клавиатуры и экрана;
3) команду запуска утилиты Norton Commander (или другой подобной утилиты).

Набрав команду, вы нажимаете клавишу < ENTER >. В результате команда поступает в интерпретатор команд MS-DOS, который поступает следующим образом. Во-первых, он определяет имя программы, которая должна быть выполнена – это последовательность символов, заканчивающаяся первым пробелом. Во-вторых, интерпретатор команд ищет адрес требуемой программы – определяет, на каком логическом диске находится загрузочный модуль программы, а также ее адрес на физическом диске.

С учетом того, что в файловой структуре системы могут находиться несколько файлов с требуемым именем, необходимо четко представлять алгоритм поиска требуемого файла, по которому работает интерпретатор команд MS-DOS:

1) во-первых, он просматривает свои внутренние таблицы. Если при этом имя файла найдено, то запускается соответствующая программа. Иначе – переход на шаг 2;
2) просматриваются все файлы текущего каталога;
3) поиск файла во всех каталогах, указанных в команде PATH файла Autoexec.bat . Например, команда PATH c:\len; c:\auto сообщает о том, что программные файлы следует искать в каталогах LEN и AUTO логического диска C: .

Если в своей команде вы не задали расширение имени файла, то в любом из трех перечисленных шагов приоритет поиска следующий:

1) ищется файл с расширением .com ;
2) с расширением .exe ;
3) с расширением .bat .

Если ни один из трех перечисленных шагов поиска не увенчался успехом, на экран выводится: **Bad command or file name** (имя команды или файла указано неверно) и вновь выдается приглашение MS-DOS.

Системные команды MS-DOS

Естественно, что перечислить все команды MS-DOS невозможно, так как имя любой программы может рассматриваться как команда. Речь может идти только о перечислении команд, требующих выполнение системных программ – утилит, лингвистических процессоров и драйверов. Соответствующая системная программа может находиться внутри MS-DOS или существовать в виде отдельного .com- или .exe-файла. Команды, соответствующие первому типу программ называются внутренними, а второму – внешними. Вот некоторые из системных команд.

Задание текущего логического диска (внутренняя команда). Для этого в ответ на приглашение MS-DOS достаточно набрать требуемое имя логического диска, например:

C: или D:

Задание текущего каталога (внутренняя команда):

CD < имя-путь каталога >

Например, в результате выполнения команды CD \SIMP\SET текущим каталогом станет дочерний каталог каталога SIMP – SET. Текущий логический диск при этом не меняется.

Вывод на экран содержимого каталога (внутренняя команда):

DIR [<имя лог. диска>][<имя[-путь] каталога или файла>][/p][/w] ,

где квадратными скобками выделены необязательные параметры.

Если параметры-имена отсутствуют, то на экран выводится содержимое текущего каталога:

DIR

Если задано имя логического диска, то выводится содержимое корневого каталога на этом диске. Например, следующая команда выводит на экран содержимое корневого каталога на логическом диске A: :

DIR A:

Если задано имя каталога, то на экран выводится его содержимое. Например, следующая команда выводит на экран содержимое каталога SIMP, являющегося дочерним каталогом по отношению к текущему каталогу:

DIR SIMP

Если задано имя файла, то на экран выводятся сведения только об этом файле. При задании имени файла разрешается вместо любой последовательности символов в имени (в том числе, и вместо расширения имени) задать символ “*”. В этом случае на экран будут выведены сведения о всех файлах, имеющих в своих именах последовательности символов, заданные в команде. Например, следующая команда выводит сведения о всех файлах текущего каталога, имеющих расширение .exe:

DIR *.exe

Если информация в каталоге слишком велика, чтобы уместиться на одном экране, то используют параметр /p . В этом случае заполнение экрана приводит к приостановке вывода до нажатия вами любой клавиши.

Параметр /w используется для сжатия выводимой на экран информации за счет того, что для каждого файла выводится лишь имя, а атрибуты (размер, дата и время создания) опускаются.

Копирование файла (внутренняя):

COPY [имя лог.диска1]<имя[-путь] файла1> [имя лог.диска2]<имя[-путь] файла2>

Данная команда создает копию файла с именем <имя файла 1> и присваивает новому файлу имя <имя файла 2>. Например, команда

COPY abc.exe c:\simp\abc.exe

копирует файл abc.exe , расположенный в текущем каталоге текущего логического диска, в файл с таким же именем, но расположенный в каталоге simp на логическом диске c: .

В качестве имени файла можно задать стандартное имя устройства ввода-вывода: prn – принтер; con – консоль (совокупность экрана и клавиатуры). Например, команда

COPY abc.txt con

выводит файл abc.txt на экран, а команда

COPY con abc.txt

позволяет ввести файл abc.txt с клавиатуры.

Команду COPY можно использовать для объединения нескольких файлов. Например, команда

COPY abc1.txt+abc2.txt abc3.txt
объединяет файлы abc1.txt и abc2.txt в файл abc3.txt .

Удаление файла (внутренняя команда):
DEL <имя файла> ,
где <имя файла> - имя удаляемого файла.

Переименование файла (внутренняя команда):
REN <имя файла 1> <имя файла 2> ,
где <имя файла 1> - старое имя файла;
<имя файла 2> - новое имя файла.

Вывод содержимого файла на экран:
TYPE <имя файла> ,
где <имя файла> - имя текстового файла (в коде ASCII).

Форматирование диска (внешняя команда – требуется утилита FORMAT.COM):
FORMAT <имя лог. диска>[/S]
Данная команда выполняет форматирование (разметку) дискеты. Для этого дискету следует вставить в дисковод A: или B: и ввести команду:
FORMAT A: (или B:) ,
в результате которой прежнее содержимое дискеты уничтожается и на ней создается пустой корневой каталог.
Задание необязательного ключа /S приводит к тому, что кроме форматирования на дискету записываются системные файлы MS-DOS. Такая системная дискета может использоваться для загрузки MS-DOS при отсутствии Windows, или если загрузчик BIOS рассматривает в качестве системного диска вначале A: (или B:), а уж затем C: (на нем находится Windows).

ПРИЛОЖЕНИЕ 2
РАБОТА В СРЕДЕ NORTON COMMANDER

Представление на экране файловой структуры

Утилита Norton Commander предназначена для того, чтобы предоставить пользователю MS-DOS удобный интерфейс для общения с этой системой. Это обеспечивается, во-первых, наглядным выводом на экран информации о файловой структуре системы. Во-вторых, Norton Commander существенно упрощает для пользователя ввод команд MS-DOS.

Для того, чтобы запустить Norton Commander, достаточно набрать команду MS-DOS – NC. Часто эту команду включают в файл Autoexec.bat и поэтому она выполняется автоматически.

В любом случае в верхней части экрана появляются две синих панели, каждая из которых содержит перечень файлов, расположенный в одном из каталогов файловой структуры системы. При этом в заголовке каждой панели указаны имя логического диска и имя каталога. Ниже располагается командная строка MS-DOS с обычным ее приглашением и мерцающим курсором. В последней строке экрана находится список клавиш <F1> - <F10> с кратким обозначением их функций.

Каталоги, отображаемые на левой и правой панелях, могут совпадать или не совпадать, но в любом случае одна из панелей, называемая активной, отображает текущий каталог. Заголовок активной панели выделяется серо-зеленым цветом. Кроме того, одно из имен файлов на активной панели выделено псевдокурсором. В отличие от обычного курсора (он находится в командной строке MS-DOS) псевдокурсор генерируется не аппаратурой, а программой (в графическом режиме экрана). Переключение активной панели производится нажатием клавиши <Tab>.

Перемещение псевдокурсора внутри активной панели производится с помощью клавиш управления курсором – вниз, вверх, влево, вправо. Нажатие клавиши <End> приводит к установке псевдокурсора на последнюю, а <Home> - на первую строку панели. Щелчком левой клавиши мыши можно установить псевдокурсор в любую позицию не только активной, но и соседней панели (происходит смена активной панели).

В общем случае панель содержит строки трех типов:

1) строку “**..**” , обозначающую выход в “родительский каталог” данного каталога;
2) строки с именами подкаталогов данного каталога (вывод прописными буквами);
3) строки с именами файлов данного каталога (вывод строчными буквами).

В последней строке панели, как правило, записано имя выделенного файла, его длина, дата и время создания или последней модификации.

Смена логического диска, соответствующего панели, производится одновременным нажатием двух клавиш: а) для левой панели - <Alt>&<F1>; б) для правой панели - <Alt>&<F2>. На экране появится диалоговое окно-меню из имен логических дисков. После этого следует установить псевдокурсор на требуемое имя и нажать клавишу <Enter>.

Для того, чтобы перейти на подкаталог текущего каталога, достаточно установить на него псевдокурсор и нажать <Enter>. Для возврата в родительский каталог требуется установить псевдокурсор на "**..**" и нажать <Enter>. Перемещаясь подобным образом вверх-вниз по файловой структуре логического диска, можно сделать текущим любой каталог на нем.

Прекращение работы Norton Commander происходит в результате нажатия клавиши <F10>. В ответ на появившийся затем вопрос о намерении прекратить работу следует нажать <Enter>.

Команды для работы с файлами

Norton Commander предоставляет пользователю команды для работы с файлами, намного более удобные, чем соответствующие команды MS-DOS. Рассмотрим их.

Создание каталога. Для этого достаточно установить в заголовке активной панели "родительский" каталог по отношению к вновь создаваемому каталогу, а затем нажать клавишу <F7>. На экране появится диалоговое окно с приглашением набрать имя нового каталога. В ответ следует набрать имя каталога прописными или строчными буквами и нажать клавишу <Enter>. В результате на активной панели появится имя нового каталога, записанное прописными буквами.

Копирование файла. Для этого требуется установить в заголовке одной из панелей "родительский" каталог по отношению к копируемому файлу, а в заголовке другой панели – "родительский" каталог по отношению к файлу-копии. Далее следует установить псевдокурсор на копируемый файл и нажать клавишу <F5>. На экране появится диалоговое окно с сообщением о готовности выполнить копирование. В ответ достаточно нажать клавишу <Enter>. (Для отмены этой или другой команды следует нажать <Esc>.) В результате на второй панели появится имя скопированного файла.

Для того, чтобы создать копию файла в том же каталоге, что и исходный файл, необходимо обеспечить, чтобы старый и новый файл имели разные имена. Для этого следует в диалоговом окне, появившемся после нажатия <F5>, набрать имя файла-копии, а уж затем нажать <Enter>.

Копирование каталога выполняется аналогично копированию обычного файла данных. При этом копируется все поддерево файловой структуры, начинающееся с данного каталога.

Копирование нескольких “дочерних” файлов (каталогов) текущего каталога можно ускорить, используя выделение группы файлов. Для выделения файла необходимо установить на его имя псевдокурсор и нажать клавишу <Ins>. В результате имя файла высвечивается желтым цветом и оно включается в группу. (Для исключения файла из группы необходимо повторить эти же два действия – установку псевдокурсора и нажатие <Ins>.) После того, как требуемая группа файлов выделена (в нижней строке панели информация о общем числе выделенных файлов и их общем объеме), ее можно скопировать последовательным нажатием <F5> и <Enter>.

Уничтожение файла. Для этого требуется установить псевдокурсор на имя уничтожаемого файла и нажать клавишу <F8>. На экране появится диалоговое окно с просьбой подтвердить намерение удалить файл. В ответ достаточно нажать <Enter> (для отмены - <Esc>).

Выделив группу файлов (аналогично выделению при копировании) ее можно уничтожить нажатием <F8> и последующими нажатиями <Enter> в ответ на вопросы Norton Commander.

Переименование файла. Для этого следует установить псевдокурсор на требуемый файл и нажать клавишу <F6>. В появившемся диалоговом окне следует набрать новое имя файла, а затем нажать <Enter>.

Создание текстового файла. Для создания нового файла необходимо одновременно нажать клавиши <Shift>&<F4>. В появившемся на экране окне необходимо набрать имя создаваемого файла и нажать клавишу <Enter>. В ответ на последующий вопрос также нажимается эта клавиша. Далее выполняется ввод с клавиатуры содержимого файла, по завершению которого следует нажать клавишу <F2>.

Редактирование текстового файла. Для работы с ранее созданным текстовым файлом необходимо установить псевдокурсор в каталоге файлов на нужный файл и нажать клавишу <F4>. Для того, чтобы переписать откорректированный файл из ОП на диск, необходимо нажать <F2>.

Ввод команд MS-DOS

В отличие от рассмотренных выше операций с файлами, для выполнения которых Norton Commander предоставляет пользователю свои команды, формат остальных команд MS-DOS остается без изменений. При этом помощь Norton Commander заключается в ускорении набора этих команд.

Крайний случай – пользователь набирает команду в командной строке MS-DOS полностью вручную, не пользуясь помощью Commander. Такой метод используется для ввода коротких или редко используемых команд.

Второй метод – для заполнения командной строки используются имена файлов, высвечиваемые на панелях. Для того, чтобы скопировать имя файла с панели в командную строку, достаточно установить псевдокурсор на требуемое имя файла и одновременно нажать две клавиши - <Ctrl>&<Enter>. При этом введенное имя файла можно редактировать точно также, как и остальную часть командной строки. Например, пусть требуется выполнить связывание нескольких объектных модулей в один загрузочный модуль. Тогда в командной строке достаточно набрать вручную лишь имя команды (TLINK), а затем перекопировать в нее по одному требуемые имена .obj-файлов.

Третий метод позволяет обойтись вообще без записи в командную строку. Установив псевдокурсор на имени исполняемого файла (расширение имени - .com, .exe или .bat), следует нажать <Enter>. Данный метод обычно применяется тогда, когда команда не имеет параметров.

Четвертый метод заключается в том, что команда MS-DOS переписывается в командную строку из протокола команд. Протокол – список последних команд (не более 16), сохраненный в Commander. Поиск нужной команды может быть выполнен двумя способами. В первом способе на экран выводится весь список (меню) команд. Для этого достаточно нажать комбинацию клавиш - <Alt>&<F8>. Установив псевдокурсор на требуемую команду в меню, следует нажать <Enter>.

Во втором способе список команд в протоколе просматривается последовательно, команда за командой. Для перехода к первой из них (самой новой) требуется нажать комбинацию <Ctrl>&<E>. При этом данная команда будет скопирована в командную строку, где она может быть скорректирована до нажатия <Enter>. Если данная команда не устраивает, то вместо нажатия <Enter> следует опять нажать <Ctrl>&<E>. В результате в командной строке окажется следующая команда из протокола. Для передвижения по протоколу на одну команду назад следует нажать комбинацию клавиш <Ctrl>&<X>.

Если программа, запускаемая любым способом из Commander, выводит какие-то данные на экран, то чтение их будет невозможно из-за присутствия на экране панелей. Для того, чтобы убрать с экрана обе панели, требуется одновременно нажать <Ctrl>&<O>. Для восстановления панелей достаточно опять нажать <Ctrl>&<O>. Удаление (восстановление) только левой панели производится одновременным нажатием <Ctrl>&<F1>, а правой - <Ctrl>&<F2>.

Настройка Norton Commander

Commander предоставляет своим пользователям возможность выполнять настройку формата информации, выводимой на экран.

В результате нажатия клавиши <F9> в верхней части экрана появляется главное (горизонтальное) меню из пяти пунктов: **Left** (левая), **Files** (файлы), **Commands** (команды), **Options** (параметры), **Right** (правая). Одна из позиций меню выделена псевдокурсором. Для выбора требуемого пункта меню достаточно установить на него псевдокурсор (с помощью клавиш управления курсором) и нажать <Enter>. Это же можно сделать щелчком левой клавиши мыши. В результате подобного выбора на экране появится ниспадающее меню, выбор в котором позволит выполнить требуемое действие.

Пункты горизонтального меню **Files** и **Commands** позволяют с помощью своих ниспадающих меню выдавать команды по работе с файлами. Многие из этих команд могут быть введены другим способом – нажатием клавиш <F1> - <F10>.

Пункты горизонтального меню **Left** и **Right** предназначены для настройки левой и правой панелей соответственно. При этом, установив в ниспадающем меню режим **Brief** (краткий), мы обеспечим вывод на соответствующей панели лишь одних имен файлов. Задание режима **Full** (полный) позволяет выводить на экран не только имена файлов, но и их основные характеристики (размер в байтах, дату и время создания или последней модификации). **Name**, **Extension**, **Time**, **Size** - группа полей в ниспадающем меню, позволяющая выполнить сортировку имен файлов, выводимых на панель, соответственно по имени, расширению имени, времени создания или модификации, размеру файла.

Пункт горизонтального меню **Options** позволяет указать информацию, выводимую вне панелей. Например, в ниспадающем меню можно установить режимы: **Path promt** (путь в приглашении) – отображение в приглашении MS-DOS имени текущего каталога; **Key bar** (строка клавиш) – отображение в нижней части экрана функциональных клавиш <F1> - <F10>; **Mini status** – вывод в нижней части каждой панели информации о выделенном файле. В этом же ниспадающем меню можно выбрать пункт **Configuration** (конфигурация). На экране появится диалоговое окно, в котором можно установить дополнительные параметры настройки Commander.