Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ
(ТУСУР)**

Кафедра компьютерных систем в управлении и проектировании (КСУП)

Е.А. Потапова

Программирование на языке ассемблера. Лабораторный практикум.

Учебное методическое пособие

2013

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данный лабораторный практикум предназначен для выполнения лабораторных работ в рамках курса «Информатика» для студентов направления 230100.62 «Информатика и вычислительная техника профиль Системы автоматизированного проектирования», для студентов направления 220400.62 «Управление в технических системах», но может быть использован при обучении студентов других направлений, предполагающих углубленную подготовку по программированию.

Цель курса - создание основы (базиса) для изучения и использования вычислительных систем в других курсах. Данная цель достигается путем обучения основам программирования на языке ассемблера для микропроцессора Intel 8086 (сокращенно - i8086).

Выбор в качестве объекта изучения именно этого языка обусловлен 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.

Данный лабораторный курс состоит из двух частей. В процессе выполнения первой части (работы 1-5) 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 6-10) производится знакомство с основами программирования на языке ассемблера. При этом объектами рассмотрения являются не только операторы ассемблера, но и методы проектирования и отладки программ.

ЧАСТЬ 1. ПРОГРАММИРОВАНИЕ НА МАШИННОМ ЯЗЫКЕ

1.1 ПРОГРАММИРОВАНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

После того как мы рассмотрели логические основы организации аппаратуры ЭВМ, перейдем к построению простейших машинных программ, выполняемых с помощью этой аппаратуры. Выполнение программ на «голой» аппаратуре будет рассматриваться нами в третьей и четвертой частях пособия, а в первых двух частях пособия все наши программы будут выполняться в среде операционной системы *DOS*. При этом для того, чтобы «не отдаляться» от аппаратуры, при построении своих программ в первой части пособия мы будем использовать единственную системную программу – *Debug*.

Debug является отладчиком, то есть программой, предназначенной для оказания помощи программистам в поиске ошибок в программах, исходные тексты которых написаны на ассемблере или каких-то других языках программирования. С помощью этой программы производится анализ и заполнение ячеек регистровой и оперативной памяти, осуществляется пошаговое выполнение программы. Преследуя цель лучше понять механизм выполнения машинных программ, мы не будем пока использовать для записи программ даже язык ассемблера, ограничившись лишь использованием вместо цифровых КОПов машинных команд соответствующих ассемблерных мнемоник.

Рассмотрим происхождение слова *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*.

Чтение и заполнение регистров

Прежде чем запустить *Debug*, необходимо запустить ту операционную систему (ОС), в среде которой будут выполняться и *Debug*, и получаемые с помощью него наши программы. В качестве такой ОС везде далее используется любая система из семейства *DOS*. Эти ОС имеют схожие пользовательские и программные интерфейсы.

Примером *DOS* является свободно распространяемая операционная система *FreeDOS*. Так как эта ОС является самостоятельной системой, выполняемой на “голой” аппаратуре, то для ее запуска из среды какой-то другой ОС, например из одной из *WINDOWS* или из *UNIX*, предварительно следует запустить программный иммитатор аппаратуры ЭВМ. Примером такого имитатора является виртуальная машина *Microsoft* “*Virtual PC*”. Пояснения об ее использовании содержатся в приложении 2. Впрочем, при выполнении первых двух частей пособия можно ограничиться запуском из *WINDOWS MS-DOS*, которая в настоящее время фактически является частью *WINDOWS*. Запуск производится:

<Пуск> → <Программы> → <Стандартные> → <Командная строка>

В результате запуска *DOS* на черном экране появится ее приглашение для ввода нами команды, например:

C:\>

Данное приглашение означает, что в данный момент времени текущим логическим диском является *C:*, а текущим каталогом – корневой каталог (\) на этом логическом диске. Далее на экране находится само приглашение – символ “>”.

З а п у с т и т е Debug, набрав его название после приглашения *DOS*, например:

C:\> *DEBUG*

Debug можно вызвать и с помощью программы *DOS Navigator*, или с помощью другой аналогичной утилиты, имеющейся на Вашей ЭВМ. Для

этого надо найти в каталоге файлов файл *Debug.com*, установить на него курсор-маркер и нажать клавишу <*Enter*>.

Дефис " _ ", который Вы видите в качестве ответа на Вашу команду – это приглашение программы *Debug*. 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 ЦП (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 MOV AL,85

Возможно, на своем экране вы увидите другие числа во второй и третьей строках. 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 MOV 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 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 команду:

D801

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

add *ax, bx*

Для того чтобы данная машинная команда была исполнена ЦП, необходимо выполнить четыре действия:

1) выбрать место в ОП для размещения машинной команды;

2) выполнить запись команды на выбранное место;

3) сообщить ЦП о том, где расположена наша машинная команда;

4) запустить ЦП для того, чтобы он исполнил команду.

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 команды. 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). Напомним, что 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*. Напомним, что регистр *CS* называется регистром сегмента кода и содержит номер параграфа, с которого начинается сегмент. Для того чтобы получить номер (реальный адрес) 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 команды 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 команды (*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) логический адрес в ОП машинной команды; 2) шестнадцатеричный код этой команды, причем младший байт команды изображен слева, а старший справа (содержимое регистров показывается наоборот – старший байт слева, а младший справа); 3) ее мнемоническое представление. Почему именно нашу команду показал *Debug*? Ответ заключается в том, что *Debug* высвечивает ту команду, младший байт которой имеет адрес: *(CS):(IP).*

Напомним, что указатель команды *IP* содержит внутрисегментное смещение младшего байта той машинной команды, которая будет исполняться следующей на ЦП. В процессе выполнения машинной программы ее команды 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 команды. *П о п р о с и т е Debug* ее выполнить, используя команду ***T*** (от "*Trace*"), которая выполняет одну команду за шаг, а затем показывает содержимое регистров. После каждого запуска *IP* будет указывать на следующую команду, в нашем случае будет указывать на *102h*. Мы не помещали никакой команды в *102h*, поэтому в последней строке распечатки мы увидим команду, 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 41 INC CX

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 команду, расположенную в памяти по адресу *l02h*, а не по адресу *l00h*.

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

Если мы захотим выполнить введенную машинную команду еще раз, то есть сложить *92Ah* и *CD1h* и сохранить новый ответ в *AX*, то надо объяснить процессору, где найти следующую команду, и чтобы этой следующей командой 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 *Debug 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 машинную команду, располагаемую в нижней части распечатки (листинга), выдаваемой командой *R*. В результате вы будете уверены, что процессор выполнит требуемую команду.

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

Напомним, что 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 машинной команде *add. В в е д и т е* два байта *00h* и *C4h*, начиная с адреса *0l00h*. Внизу листинга регистров вы увидите команду "*add ah,al*", которая суммирует два байта регистра *AX* и записывает 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*.

Вычитание двух чисел

Мы собираемся написать машинную команду 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* команду вычитания в ОП:

_E 100

3756:0100 01.29 D8.D8

Листинг регистров (не забывайте установить *IP* в *100h*) должен теперь показать команду "***sub*** *ax,bx*", которая вычитает содержимое регистра *BX* из регистра *AX* и записывает результат в *AX*.

В ы п о л н и т е эту машинную команду с помощью команды *Debug T*. *AX* должен содержать *CD1*. Измените *IP* так, чтобы он указывал на эту машинную команду, и выполните ее опять (не забывайте сначала проверить команды внизу листинга регистров), *AX* теперь должен содержать *03A7h*.

И с п о л ь з у й т е машинную команду *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).

Умножение двух чисел

Команда 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 машинная команда *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* команду умножения *E3F7h* по адресу *0l00h* и установите *AX=7C4Bh* и *BX=100h*. Вы увидите команду 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

машинной команде, на регистр *AX*, и сохраняет ответ в паре регистров *DX:AX*.

Перед запуском команды 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 машинной команды умножения. 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 машинную команду *F3F7h* по адресу *0100h* (и *101h*). Как и команда *mul,* команда деления ***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 команды деления по адресу *0l00h* мы получим для *AX=7C4Bh* результат нашего деления и для *DX=0012h* остаток.

Лабораторная работа 1

В ходе выполнения лабораторной работы требуется выполнить с помощью *Debug* пошаговое вычисление выражения:

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

где *X1 – X5* - десятичные целые числа, взятые в соответствии с номером вашего варианта *K* из таблицы 2.

Т а б л и ц а 2. Исходные числа для лабораторной работы 2

K	*X1*	*X2*	*X3*	*X4*	*X5*	*K*	*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 в виде экранного изображения показывается преподавателю. Одновременно, преподавателю показывается на листе бумаги подробный ручной расчет заданного выражения в десятичной системе счисления. При этом итоговое число должно быть записано в виде обыкновенной дроби, например, $17\frac{8}{27}$.

1.2 ВЫВОД СИМВОЛОВ НА ЭКРАН

В предыдущем разделе мы выполняли с помощью *Debug* запись в ОП и последующее выполнение всего одной машинной команды, производящей над своими операндами какую-то арифметическую операцию. Теперь пришла пора заняться получением программы, состоящей из нескольких команд. Как и прежде, первая команда программы будет получать управление из *Debug*. Кроме того, последняя команда нашей программы должна возвращать управление обратно в *Debug*. В процессе своего выполнения программа будет обращаться за помощью к *DOS* с целью вывода

на экран строки символов. Единственным способом вызова из прикладной программы системных подпрограмм *DOS* или *BIOS* является размещение в этой программе машинной команды программного прерывания. Термин "прерывание" означает, что выполнение нашей программы прерывается (приостанавливается) на время, необходимое для выполнения требуемой системной программы. Команда программного прерывания обозначается как ***int*** (от "*Interrupt*" – прерывание). Команда *int* для функций *DOS* имеет вид "*int 21h*", в машинном коде *21CDh*.

Вывод одного символа

Примером функции *DOS*, выполнение которой мы можем запросить из программы с помощью команды "*int 21h"*, является вывод символа на экран. Для того чтобы различать функции *DOS*, которых много, используется регистр *AH*. При выводе одного символа в него помещается *02h*. В регистр *DL* заносится код *ASCII* выводимого символа. В табл. 3 приведены отображаемые (видимые на экране) коды *ASCII*.

Таблица 3. 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

Допустим, что мы хотим вывести символ *A*, тогда в регистр *DL* мы должны поместить число *41h*. *П о д г о т о в ь т е* регистры и память для последующего выполнения команды *“int 21h”*. 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 машинной команды 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 команд. Пошаговое выполнение этой подпрограммы вам скоро наскучит. Но если вы не доведете его до конца, то ваш компьютер "зависнет".

Но если вы все-таки, 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 команда 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-смещение, 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 вывел на экран букву *A* и возвратил затем управление в нашу программу. (Машинная команда, размещенная по адресу *102h*, осталась от другой программы, поэтому последняя строка вашего листинга может выглядеть по-другому.)

Команда завершения программы

Машинная команда "*int 20h"* сообщает *DOS* о том, что мы хотим выйти из нашей программы, и чтобы управление опять вернулось в *DOS*. В нашем случае эта команда вернет управление в *Debug*, так как мы запускаем нашу программу не непосредственно из *DOS*, а из *Debug*.

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

_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 *Debug* выполняет всю программу (сейчас она состоит всего из

одной команды – “*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 машинную команду “*int 20h”* в конец любой программы для того, чтобы красиво передать управление *DOS* (или *Debug*). Для начала поместим ее после команды “*int 21h”* и получим программу из двух команд, выполняющую вывод символа на экран. Для этого начиная с адреса *l00h в в е д и т е* одну за другой две машинные команды – *21CDh* и *20CDh*.

Когда у нас была только одна машинная команда, то мы могли "пролистать" ее командой *R Debug*, но теперь у нас две команды. Чтобы увидеть их, воспользуемся командой *Debug* ***U*** (от "*Unassemble*" – разассемблирование):

_U 100

3970:0100 CD21 INT 21

3970:0102 CD20 INT 20

Далее идут еще 12 строк листинга, содержащие команды, 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 команды 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) машинные команды. 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 машинные команды в мнемонической форме, а число 100 в команде означает, что ввод машинных команд 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 команда самой программы – ***mov***. Эта же команда 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 машинную команду "*mov ah,dl"*. Эта команда 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*.

Команда *mov* пересылает число из второго регистра в первый, и по этой причине мы пишем *AH* перед *DL*. Машинный код данной команды *D488h*. Существуют другие формы этой же команды *mov*. Они имеют другие машинные коды и выполняют другие операции пересылки. Например, следующая команда (*C389h*) выполняет пересылку не байтов, а слов между двумя регистрами *AX* и *BX*:

3970:0100 89C3 MOV BX, AX

Следующая форма команды *mov* записывает значение числа в регистр, не используя другой регистр-источник:

3970:0100 B402 MOV AH, 02

Эта команда загружает число *02h* в регистр *AH*. Старший байт команды, *02h* является числом, которое мы хотим загрузить**.** *З а п и ш и т е* эту

команду 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 команды *mov* для того, чтобы установить регистры *AH* и *DL* перед выполнением команды "*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 машинная команда в нем занимает два байта (в общем случае это не выполняется). У нас четыре команды, 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 файл *Writestr.com* находится в корневом каталоге (\) на диске "*C:"*, и его длина составляет восемь байт. Чтобы загрузить и выполнить программу, наберите в ответ на приглашение *DOS writestr.com* 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*, выводит на экран целую строку и прекращает вывод, когда находит символ "***$***".

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

48 65 6C 6C 6F 2C 20 44

4F 53 20 68 65 72 65 2E

24

Последнее число *24h* является *ASCII*–кодом для символа *$*, и оно сообщает *DOS*, что это конец строки символов. Теперь посмотрим, что содержит 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 машинных команд. Подобно *U* поместите после *D* адрес, чтобы сообщить *Debug*, откуда начинать дамп.

Н а б е р и т е команду "*D 200*". Она выведет 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. 4.

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.

Лабораторная работа 2

Требуется разработать программу, которая выводит на экран ваши имя и фамилию, записанные сначала русскими, а затем английскими буквами. Программа должна вызываться непосредственно из *DOS*.

Т а б л и ц а 4. 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*

1.3 ВЫВОД НА ЭКРАН ДВОИЧНЫХ ЧИСЕЛ

Приступим к решению задачи отображения на экран двоичных чисел, содержащихся в ячейках памяти, называемых регистрами. Напомним, что в ячейке любой памяти (в регистровой, ОП, ВП) любая информация содержится в виде последовательности битов. Поэтому отображение этой информации на экране в виде двоичного числа имеет практический смысл.

В ходе решения указанной задачи нам потребуется рассмотреть новые машинные команды, а также произвести знакомство с новыми понятиями. При этом мы начнем с более близкого знакомства с одним из битов рассмотренного ранее регистра *FLAGS* – флага переноса.

Флаг переноса

Если выполнить сложение чисел 1 и *FFFFh*, то получим *10000h*. Это число не может быть записано в шестнадцатибитное слово, так как 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". Напомним, что флаг *CF* является одним из шестнадцати битов в регистре флагов *FLAGS*.

В ы п о л н и т е загрузку чисел 1 и *FFFFh* в регистры *BX* и *AX* и запишите в память команду "*add ax,bx"*. После этого протрассируйте эту команду. 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 команды 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 выводим 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, а затем повторим этот процесс для каждой последующей цифры, то во флаге переноса будут по очереди содержаться все цифры нашего двоичного числа.

Команда ***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. На рис.12 показано наглядное представление работы команды *rcl*.

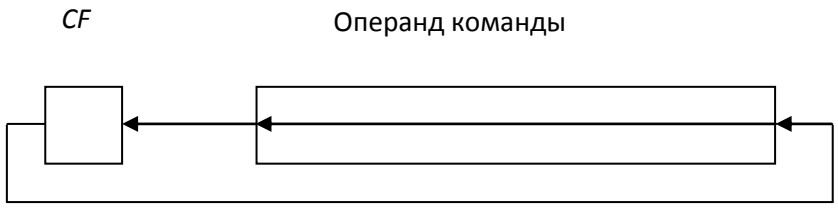


Рис. 12. Команда циклического сдвига влево через перенос *rcl*

В ы п о л н и т е с помощью *Debug* размещение по адресу *100h* команды “*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 команду несколько раз. Убедитесь, что после 9 циклов регистр *BX* содержит опять *B7h*.

Как вывести на экран 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 команду ***adc*** (от “*Add with Carry*” – сложение с переносом). Эта команда складывает три числа: два числа, как и команда *add*, а также один бит из флага переноса.

П о м е с т и т е в память после команды “*rcl bl,1”* команду “*adc dl,30* “, которая выполнит сложение содержимого *DL* (0), *30h* и флага переноса, поместив результат в *DL*. Записав далее команды, обеспечивающие вывод символа на экран и завершение программы, получим программу, выполняющую вывод на экран старшего бита регистра *BL*:

```
mov     dl, 00
rcl             bl, 1
adc     dl, 30
mov     ah, 02
int     21
int     20
```

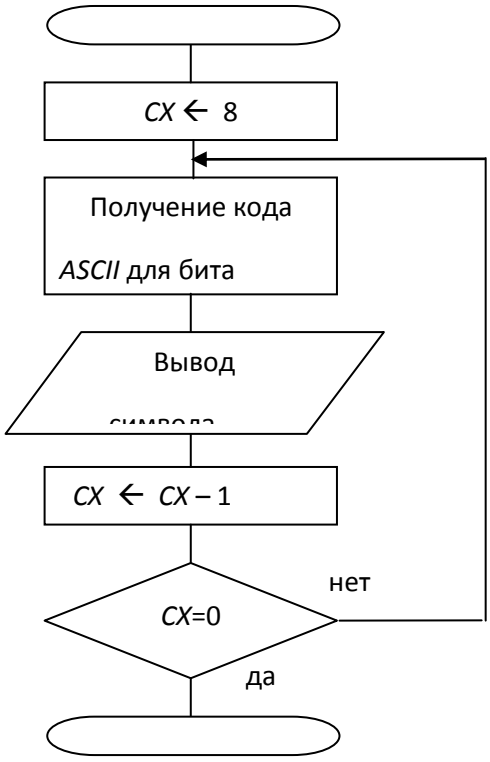
В ы п о л н и т е эту программу для обоих значений старшего бита *BL*. Для записи в *BX* используйте команду *R Debug*.

Организация циклов

Если мы хотим вывести на экран 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.13. 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.13 это этап “Завершение алгоритма”.



CX - счетчик повторений

Рис. 13. Алгоритм вывода на экран содержимого байта

Для организации циклов используются специальные машинные команды. Одной из них является команда ***loop***. Она записывается в конце цикла, т.е. после тех команд, выполнение которых следует повторить. Данная команда имеет один операнд – адрес первой из повторяемых машинных команд. 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.

Выполнение команды *loop* сводится к следующему. Во-первых, она вычитает из содержимого регистра *CX* единицу. Во-вторых, она сравнивает полученное содержимое регистра *CX* с нулем, и если оно не равно 0, то делает переход по адресу, заданному в качестве операнда команды *loop*. В-третьих, если переход не делается, то на ЦП начинает выполняться машинная команда, расположенная в программе сразу же за командой *loop*. Таким образом, наличие данной команды обеспечивает реализацию двух этапов алгоритма, приведенного на рис.13.

В качестве примера применения команды *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 вспомнить, что не следует использовать команду *T Debug* для команды *int*. При достижении этой команды следует набрать команду *Debug “G d”*, где *d* – адрес в памяти команды, следующей за *int*. При достижении команды “*int 20”* вводится команда *G Debug*.

При применении команды “*G d”* необходимо знать адрес останова *d*. Исключить трассировку при достижении машинной команды *int* проще, если использовать команду *Debug* ***P*** (от "*Proceed*" – переходить, продолжать). Эта

команда является удобным средством обхода команд *int*, 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.13).

Отладка программы

Отладка программы включает поиск ошибок (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. При этом заметим, что *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.

Лабораторная работа 3

Требуется разработать программу, вызываемую из *DOS*, которая за одно свое выполнение выводит на экран двоичное содержимое двух заданных регистров. Для выбора регистров используйте таблицу 5. Пример сообщения, выводимого программой на экран:

(AX) = 1010101011111111 (BX) = 0111100100111001

Т а б л и ц а 5. Задание регистров

K	1	2	3	4	5	6	7	8	9	10
Рег.1	*AX*	*BX*	*CX*	*DX*	*BP*	*SI*	*DI*	*BX*	*CX*	*DX*
Рег.2	*BX*	*CX*	*AX*	*BX*	*AX*	*BX*	*CX*	*DX*	*BP*	*BP*
K	11	12	13	14	15	16	17	18	19	20
Рег.1	*BP*	*SI*	*DI*	*AX*	*BX*	*CX*	*DX*	*BP*	*SI*	*DI*
Рег.2	*DI*	*AX*	*DX*	*DX*	*DI*	*DX*	*SI*	*BX*	*CX*	*BP*

Примечание 1. Так как программа вызывается не из *Debug*, а из *DOS*, то она сама должна задавать первоначальное содержимое регистров, выводимых на экран. Для этого рекомендуется использовать команды *mov*.

Примечание 2. Если регистр, содержимое которого выводится на экран, используется в вашей программе и для других целей (например, *CX* – счетчик цикла), то первоначальное содержимое этого регистра необходимо переписать с помощью команды *mov* в какой-то другой, свободный регистр данных и выполнять вывод на экран содержимого этого регистра.

1.4 ВЫВОД НА ЭКРАН ЧИСЕЛ В ШЕСТНАДЦАТЕРИЧНОЙ ФОРМЕ

В ходе рассмотрения предыдущего раздела мы научились выводить на экран двоичное содержимое регистров. Аналогично делается вывод на экран и двоичного содержимого ячеек ОП. Но гораздо удобнее для человека увидеть содержимое этих же ячеек не в двоичной, а в шестнадцатеричной системе. Напомним, что шестнадцатеричное число фактически есть сокращенная форма записи двоичного числа. При этом каждая шестнадцатеричная цифра соответствует четырем двоичным цифрам, то есть описывает содержимое четырех битов памяти.

В ходе рассмотрения данного вопроса производится практическое знакомство с тремя флагами состояния, командами для работы с этими флагами, а также с некоторыми другими важными машинными командами.

Флаги состояния

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

Допустим, что мы выполнили машинную команду вычитания *sub*. Одним из флагов состояния, устанавливаемых в зависимости от результата этой команды, является ***флаг нуля ZF*** ("*Zero Flag*"). Если результат команды *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 нуль).

В в е д и т е в память команду “*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 SF ("*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 команды "*sub ax,bx"* при разных содержимых *AX* и *BX*, и наблюдая за флагом знака.

Флаг переполнения OF 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 -2288). Это ошибка, так как результат переполняет слово. На листинге регистров переполнение обозначается как *OV* ("*Overflow*" – переполнение). Если предыдущая арифметическая команда 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 команду *sub* или *add*.

Использование команды *sub* для сравнения двух чисел неудобно, так как эта команда производит изменение первого из чисел. Другая команда, ***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 команду "*cmp ax,bx"*. При этом убедитесь, что установлен флаг нуля (*ZR*), но оба регистра сохранили свое значение – *F5h*.

Команды условного перехода

Напомним, что 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 команды, называемые командами ***условного перехода***.

Команда ***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 командой *sub* напишем, например,*"jz 15a"*, то нулевой результат вычитания означает, что ЦП начнет выполнять не следующую по порядку команду, а команду, находящуюся по адресу *15A*.

Противоположной по отношению к *jz* является команда *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 команды введите команду *G Debug*.

Команда условного перехода ***ja*** (от "*Jump if Above*" – перейти, если больше) осуществляет переход на указанный в команде адрес, если по результатам предыдущей команды флаг переноса *CF* сброшен (на листинге *CF = NC*). Флаг нуля *ZF* также должен быть сброшен. Данная команда обычно записывается сразу за командой сравнения (*cmp*) двух беззнаковых чисел. Если первое сравниваемое число больше второго, то команда *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, его можно вывести на

экран. *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.14 приведена блок-схема программы, выполняющей вывод на экран 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*. Команда *cmp* вычитает два числа ((*DL*)–*9h*), чтобы установить флаги, но она не изменяет регистр *DL*. Поэтому, если содержимое *DL* больше, чем 9, команда "*ja 10c"* осуществляет переход к команде 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 Debug* с указанием точки останова, или команду *P*, когда запускаете машинную команду *int*. Затем проверьте правильность работы программы, используя команду *G***,** предварительно загружая в *BX* граничные данные: 0; 9; *Ah* и *Fh* **.**

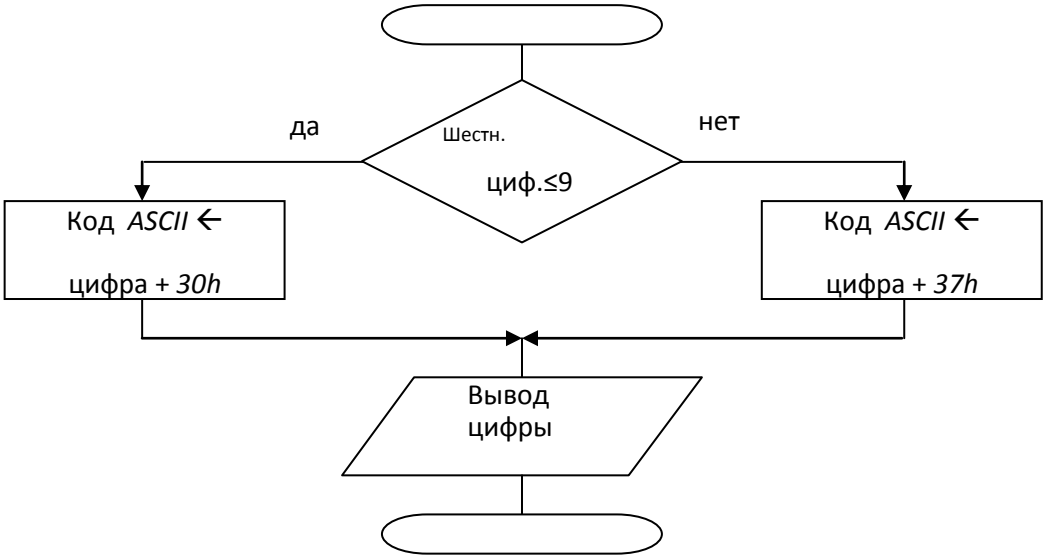


Рис. 14**.** Алгоритм программы вывода одной шестнадцатеричной цифры

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.15. При этом детальный алгоритм каждого из двух этапов “Вывод цифры на экран” приведен ранее на рис.14. Рассмотрим этап "Выделение старшей цифры". В результате него содержимое старшего полубайта должно быть переписано (сдвинуто) в младший полубайт.

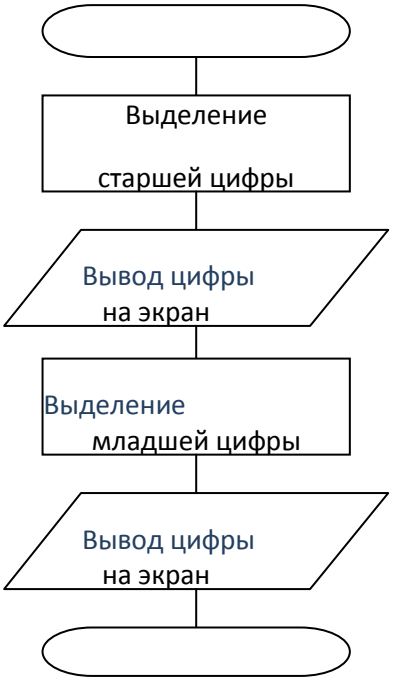


Рис. 15. Алгоритм вывода двузначного шестнадцатеричного числа

Несмотря на то, что нам надо выполнить сдвиг вправо, вспомним команду *rcl*, которая циклически сдвигает байт или слово влево через флаг переноса. Ранее мы использовали команду “*rcl bl,1”*, в которой единица есть сообщение для ЦП о том, что надо сдвинуть содержимое *BL* на один бит. Мы можем осуществить циклический сдвиг более чем на один бит, но мы не можем написать команду “*rcl bl,2”*. Для циклического сдвига необходимо поместить счетчик сдвигов в регистр *CL*, который используется здесь так же, как регистр *CX* применялся командой *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 команду сдвига ***shr*** (“*Shift Right*" – логический сдвиг вправо). Данная команда не только выполняет сдвиг вправо, но и записывает в освобождающиеся старшие биты нули. В этом проявляется разница между терминами "логический" и "циклический", так как команда 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 команду сдвига:

```
100     shr     dl,cl
```

DL должен теперь содержать число *05h*. То есть этот регистр содержит в своем младшем полубайте старшую цифру числа *5Dh*.

Реализацию этапа "Выделение старшей цифры" осуществляют команды:

```
mov   dl,bl
mov   cl,04
shr   dl,cl
```

П о м е с т и т е эти команды в ОП, дополнив их командами этапа "Вывод цифры на экран". При этом не забудьте скорректировать адреса переходов. (Можно записать программу со старыми адресами, а затем скорректировать команды 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 команду ***and*** (логическое "*И*"). Данная команда 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, команда "*and bl,cl"* последовательно выполняет операцию *and* сначала над битами 0 регистров *BL* и *CL*, затем над битами 1, битами 2 и так далее, 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 команды 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 команду "*int 20"*.)

Лабораторная работа 4

Требуется разработать программу вывода на экран двух четырехзначных шестнадцатеричных чисел, содержащихся в заданных регистрах (те же регистры, что и в работе 4).

Пример сообщения, выводимого программой на экран:

(AX) = 67FE (BX) = BA59

При написании программы учтите примечания к работе 4, а также следующие:

Примечание 1. Некоторые 8-битные регистры следует заменить 16-битными.

Примечание 2. При реализации вывода второй и третьей цифр числа сдвигу числа вправо должен предшествовать его сдвиг влево. Для выполнения сдвига влево используйте команду ***shl*** ("*Shift Left*" – логический сдвиг влево). Использование этой команды 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*.

1.5 ВВОД С КЛАВИАТУРЫ ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

В предыдущем разделе были рассмотрены теоретические вопросы, связанные с использованием в программе стека и процедур. Теперь перейдем к практическому рассмотрению этих вопросов в процессе решения важной задачи ввода с клавиатуры шестнадцатеричных чисел. Попутно рассмотрим некоторые новые машинные команды.

Ввод одной шестнадцатеричной цифры

Для того чтобы ввести с клавиатуры в программу *ASCII*-символ, можно воспользоваться командой программного прерывания "*int 21h*" с функцией номер 1. Вызванная в результате данной команды подпрограмма *DOS* помещает *ASCII*-символ, соответствущий нажатой клавише, в регистр *AL*.

В в е д и т е команду "*int 21h*" по адресу *l00h*, поместите номер функции 1 в регистр *AH*, а затем запустите команду 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.30 приведена блок-схема программы, которая выполняет ввод цифры с клавиатуры в регистр *AL*.

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 команду условного перехода *ja* (перейти, если больше), но и обратную ей команду ***jbe*** (перейти, если меньше или равно). Обе команды 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 команда "*int 20*" (*Debug* восстанавливает регистры после этой команды). Поэтому для запуска программы используйте команду *Debug* "*G d*", где *d* – смещение команды "*int 20*".

В ы п о л н и т е 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.

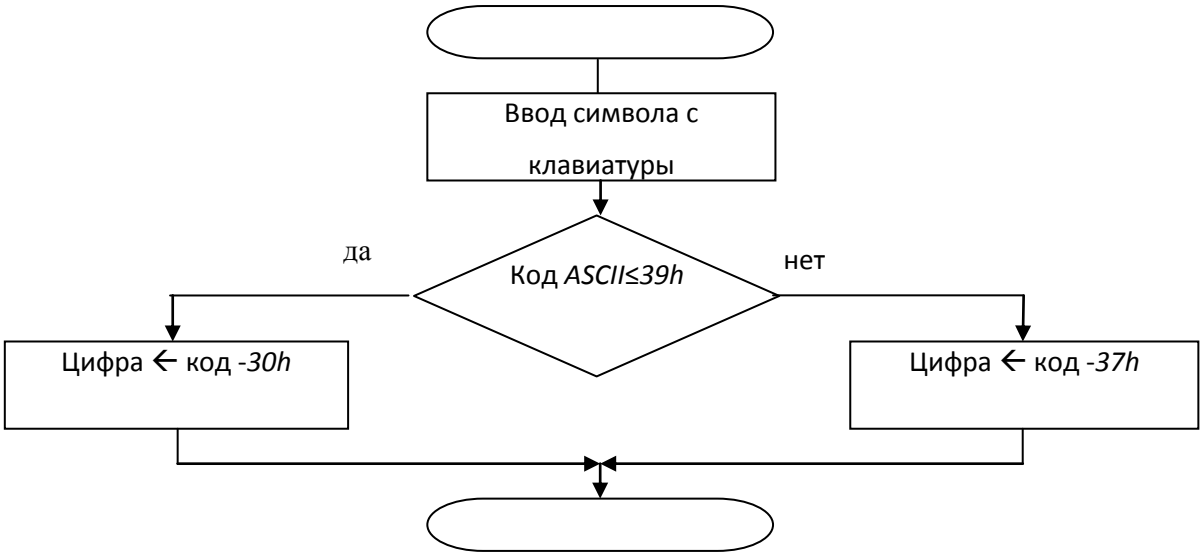


Рис. 30. Алгоритм ввода шестнадцатеричной цифры

Ввод двухзначного 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.31.

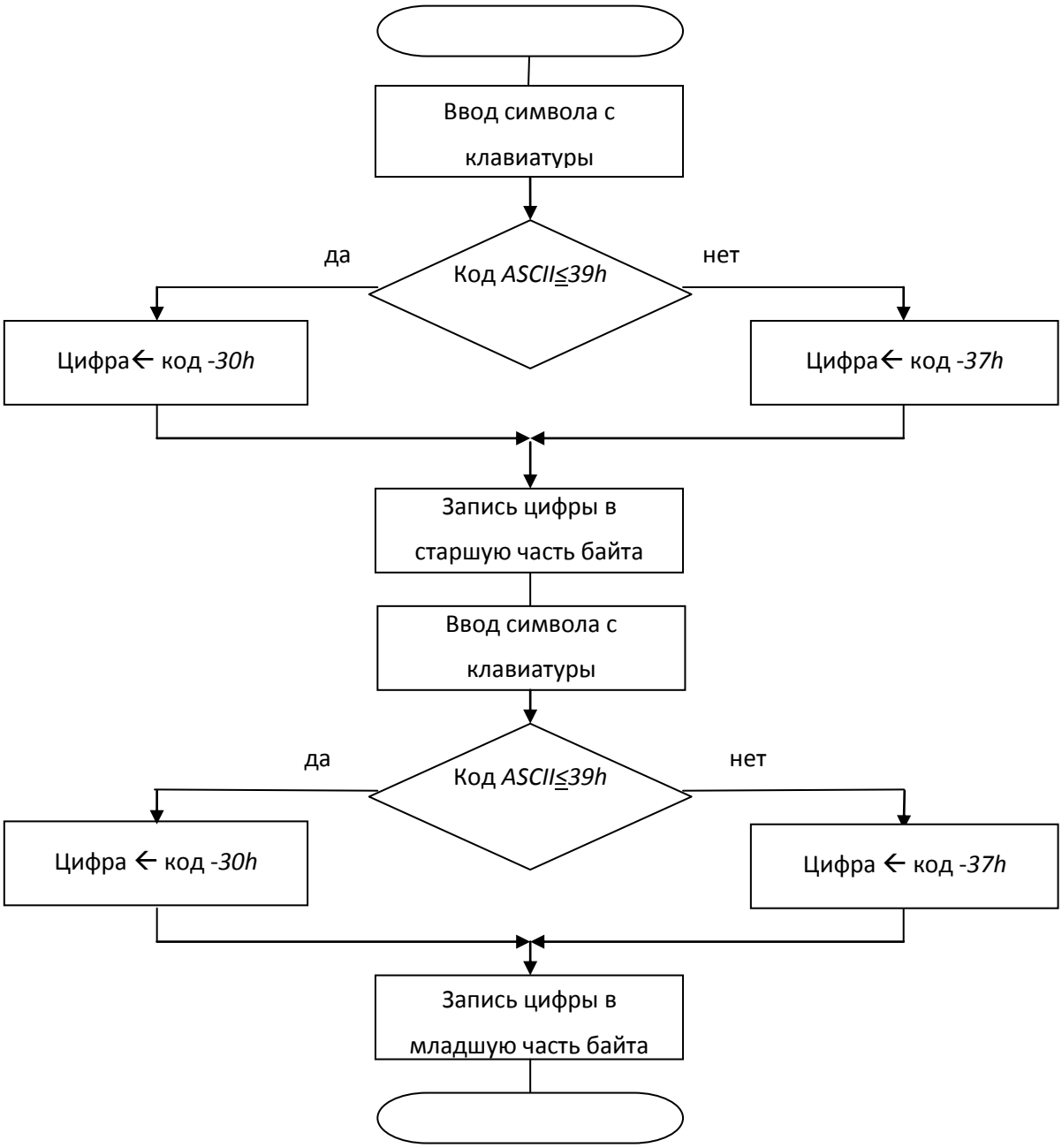


Рис.31. Алгоритм ввода двузначного шестнадцатеричного числа

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 команду *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 команды "*int 20h*". (Для ввода шестнадцатеричных чисел используйте только заглавные буквы.)

Более совершенный ввод шестнадцатеричных цифр

Записанные ранее программы ввода одно- и двухзначного шестнадцатеричных чисел реагировали на нажатие "нецифровых" клавиш точно так же, как и "цифровых". Теперь мы получим программу, которая не будет реагировать на нажатие "нецифровых" клавиш.

Для этого в состав программы введем 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.32 приведена блок-схема процедуры ввода шестнадцатеричной цифры. Текст этой процедуры (она возвращает цифру в регистре *BL.*):

```
200         push  ax
201         push  dx
```

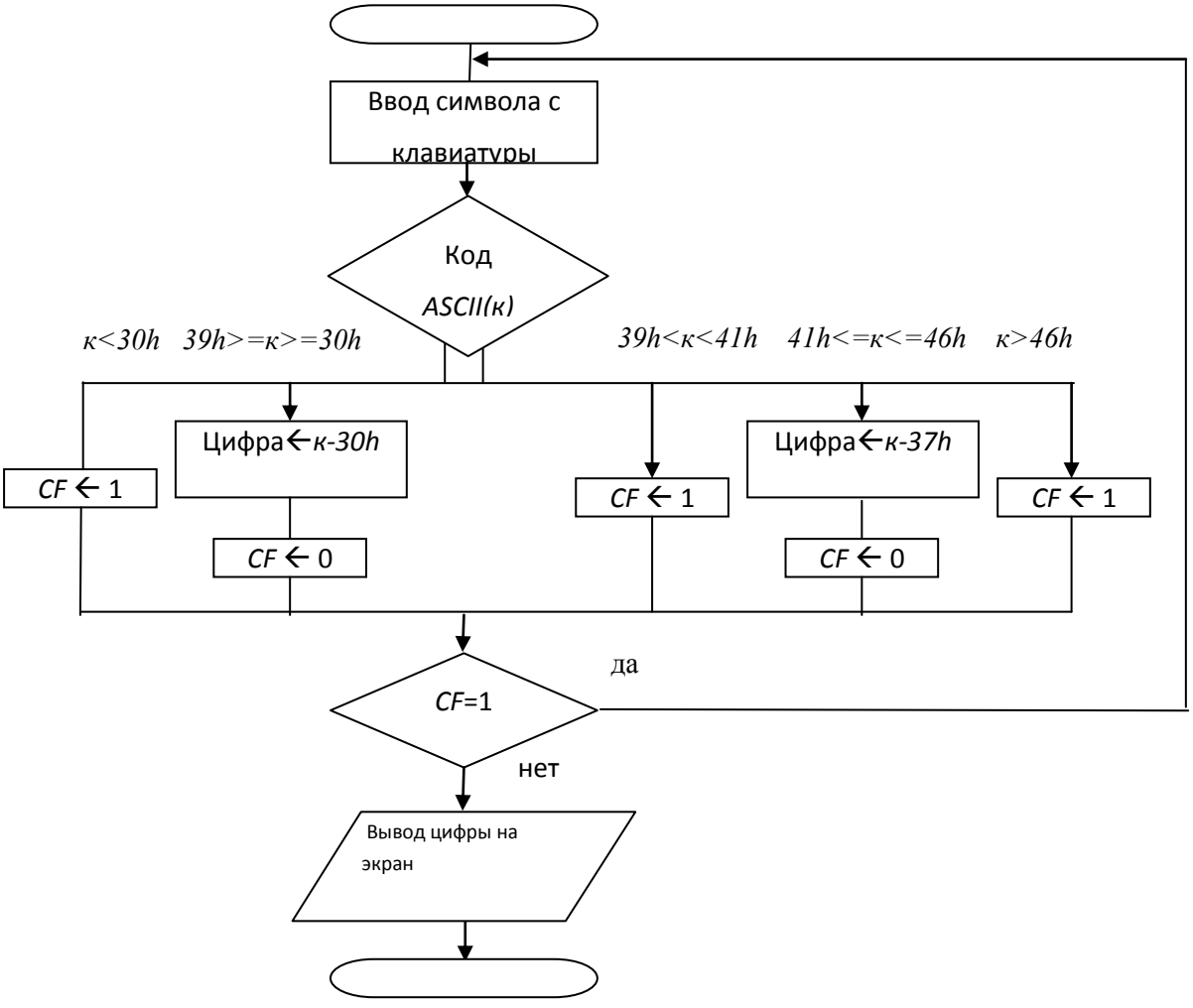
```
202         mov   ah,8
204         int   21
206         mov   dl,al
208         cmp   al,30
20A         jb    222
20C         cmp   al,39
20E         ja    215
210         sub   al,30
212         clc
213         jmp   223
215         cmp   al,41
217         jb    222
219         cmp   al,46
21B         ja    222
21D         sub   al,37
21F         clc
220         jmp   223
222         stc
223         jc    204
225         mov   bl,al
227         mov   ah,2
229         int   21
22B         pop   dx
22C         pop   ax
22D         ret
```

И з у ч и т е внимательно алгоритм и найдите реализацию его этапов в тексте процедуры. 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*.

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 команды для работы с флагом *CF*. Команда ***stc*** устанавливает флаг (*CF=1*), а ***clc*** сбрасывает его (*CF=0*). Команда условного перехода ***jc*** выполняет переход при *CF=1*, а команда ***jnc*** – при *CF=0*.



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

Рис. 32. Алгоритм ввода одной шестнадцатеричной цифры

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 Debug* для перехода через команды *int*. Курсор появится в левой части экрана и будет ждать ввода символа. Напечатайте символ "*K*", который не является правильным символом. Ничего не произойдет. Теперь напечатайте один из заглавных шестнадцатеричных символов. Вы должны увидеть шестнадцатеричную цифру в регистре *BL*, а также на экране. Испытайте эту процедуру на граничных условиях: "\" (символ, стоящий перед нулем), "0", "9", ":" (символ, стоящий после 9) и так далее.

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.33.

Рис. 33. Алгоритм ввода двузначного шестнадцатеричного числа

Лабораторная работа 5

Требуется дополнить разработанную в работе 4 программу вывода на экран двоичного содержимого двух регистров так, чтобы первоначальное содержимое этих регистров вводилось с клавиатуры в виде шестнадцатеричных чисел.

Пример информации на экране:

ВВЕДИТЕ СОДЕРЖИМОЕ РЕГИСТРА AX F46B
ВВЕДИТЕ СОДЕРЖИМОЕ РЕГИСТРА BX 5A0C
(AX) = 1111010001101011 (BX) = 0101101000001100

Примечание 1. Рекомендуется дополнительно разработать процедуры, одна из которых выполняет ввод шестнадцатеричного числа в 16-битный регистр, а другая – вывод содержимого регистра в двоичном виде.

Примечание 2. Следует обратить особое внимание на недопустимость использования одного и того же регистра одновременно для нескольких целей. При этом для временного хранения содержимого регистра удобно использовать стек. Например, с помощью команды "*push ax*" можно записать прежнее содержимое *AX* в стек, затем использовать регистр *AX* для других целей, а затем вернуться к прежнему содержимому этого регистра с помощью команды "*pop ax*". (При этом если удобно, слово из стека можно "вытолкнуть" не в *AX*, а в любой другой регистр.)

Примечание 3. Для того чтобы выполнить перевод экранной строки, следует последовательно вывести на экран два символа (в любом порядке): *0ah* (перевод строки) и *0dh* (возврат каретки).

ЧАСТЬ 2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

2.1 ПРОСТЫЕ ПРОГРАММЫ НА АССЕМБЛЕРЕ

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 команду, 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:

```
[org 100h]
. . . . . .
int     20h
```

Посмотрим внимательно на данную структуру. По горизонтали она разделена на две части. Левая (пока пустая) часть предназначена для записи идентификаторов (меток) 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 команды программы. При этом 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.

В конце программы будем помещать хорошо знакомый нам исполнительный оператор “*int 20h*”, выполняющий возврат из программы туда, откуда она была запущена.

Пример программы на ассемблере

В качестве примера запишем на ассемблере ту программу *Writestr*, которую мы создали в п. 4.3 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:

```
[org   100h]
mov   ah,2h
mov   dl,2ah
int    21h
int    20h
```

Следует отметить, что если в программе есть шестнадцатеричные 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 или на любом другом языке программирования 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*. Примерами такого редактора является *Edit*. , запускаемый из командной строки *DOS*, или текстовый редактор в *DOS Navigator*. *С о з д а й т е* исходный файл *Writestr.asm*, поместив в него приведенную выше программу на ассемблере. 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. Теперь давайте начнем ассемблировать программу *Writestr*:

C:\ > NASM Writestr.asm –o Writestr.com

Ответного сообщения транслятора в случае успешной трансляции не будет:

C:\ >

В результате транслятор-ассемблер создал файл, называющийся *Writestr.com*, который вы найдете на диске. Конечная часть команды “*-o Writestr.com*” используется для задания имени результирующему файлу (*Writestr.com*). При отсутствии этой части результирующий файл получит имя *Writestr*.

Н а п е ч а т а й т е "*Writestr.com*", чтобы запустить *com*-файл 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 3.

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 команды. Но в отличие от псевдооператоров они не интересуют и сам транслятор, существуя лишь для человека, который читает программу.

Добавим комментарии в записанную ранее программу на ассемблере:

```
        [org   100h]
;       Вывод звездочки на экран
;       ---------------------------------
        mov   ah,2h       ; Функция вывода символа
        mov   dl,2ah      ; Символ * в DL
        int   21h         ; Вывод символа
        int   20h         ; Возврат в DOS
```

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 функций 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 команд 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* команды переходов содержали адреса тех команд программы, на которые переход делался. Допустим, что мы решили добавить в программу новые команды. 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) латинские 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. В отличие от операционных систем *WINDOWS, DOS*, а также многих других программ, транслятор-ассемблер *NASM* различает заглавные и строчные буквы в именах меток. Для него это совершенно разные символы. Примеры меток: *count, Page25, $E10*. Весьма желательно, чтобы метки поясняли смысл программы, а не выбирались отвлеченно.

Метки бывают локальными и глобальными. ***Глобальная метка*** может использоваться во всем исходном файле. Имя такой метки не может начинаться с точки. Имя ***локальной метки*** начинается с точки. Областью действия такой метки является фрагмент исходной программы, заключенный между двумя ближайшими глобальными метками. Вне этого фрагмента может быть задана другая локальная метка с точно таким же именем.

Особенно полезно применение локальных меток внутри процедур. Дело в том, что в разных процедурах часто используются идентичные метки. В процессе сборки исходной программы транслятором (такая сборка производится при обработке рассматриваемого позже псевдооператора *include*) он выведет сообщения об ошибках только в том случае, если идентичные метки являются глобальными. Если эти метки оформлены как локальные, то никаких ошибок не будет, так как это совершенно различные метки. Заметим, что использование идентичных меток внутри процедур очень удобно для программиста, позволяя присваивать схожим фрагментам процедур одинаковые имена. Пример: использование метки “*.Exit*” для заключительного фрагмента процедуры.

Некоторые из буквенных слов зарезервированы транслятором-

ассемблером и не могут быть использованы в качестве меток. Сюда относятся имена регистров (например, *AX*), мнемоники исполнительных команд (например, *add*), указатель на точку входа (*..start*), и псевдооператоры, например, "*end"*.

Еще один пример программы

Ранее в п. 7.4 мы создали небольшую программу, которая выводила на экран 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:

```
                [org  100h]
;           Вывод 10 символов от A до J
;           ------------------------------------
;   Вызовы:   Write_char
Start:
                mov   dl,'A'        ; В  DL  код  буквы  A
                mov   cx,10         ;В счетчике символов -10
.Loop:          call  Write_char    ; Вывод символа
                loop  .Loop         ; Переход к следующему символу
                int   20h           ; Возврат в  DOS
;       Вывод символа на экран и увеличение кода символа на 1
;       ------------------------------------------------------------------------
```

```
;   Входы:     DL содержит код символа
;   Выходы:    DL содержит код нового символа
Write_char:
                push  ax
                mov   ah,02       ; Функция вывода символа
                int   21h         ; Вывод символа на экран
                inc   dl          ; Код следующего символа
                pop   ax
                ret               ; Возврат из процедуры
```

Обратите внимание, что в первом операторе программы 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, оператор "*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.

Вывод на экран двузначного шестнадцатеричного числа

В п. 6 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. 39 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_byte_hex*. Данная процедура предназначена для тестирования процедуры *Write_byte_hex*, выполняющей вывод на экран двузначного шестнадцатеричного числа, содержащегося в байте. В свою очередь, процедура *Write_byte_hex* вызывает (инициирует) процедуру *Write_digit_hex*, выполняющую вывод на экран шестнадцатеричной цифры. В ходе своей работы данная процедура вызывает процедуру *Write_char*, выполняющую вывод символа на экран.

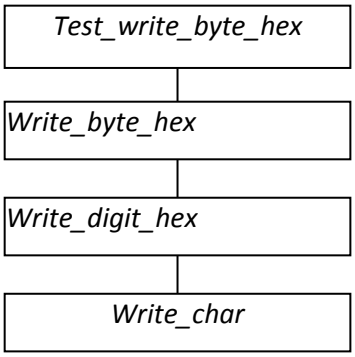


Рис. 39. Дерево подпрограмм для программы вывода двузначного шестнадцатеричного числа

В п. 6 приведены блок-схемы для процедур *Write_byte_hex* и *Write_digit_hex*. В следующей программе тексты этих процедур опущены:

```
                [org   100h]
;          Тестирование процедуры  Write_byte_hex
;          ----------------------------------------------
;   Вызовы:   Write_byte_hex
Test_write_byte_hex:
                mov   dl,3fh              ; Тестировать с 3Fh
```

```
                call    Write_byte_hex     ; Вывод числа
                int     20h                ; Возврат в DOS
;               Вывод  двузначного шестнадцатеричного числа
;               -----------------------------------------------------
;   Входы:  DL  содержит  выводимое  число
;   Вызовы:   Write_digit_hex
Write_byte_hex:                            ; Тело процедуры Write_byte_hex
                ret
;           Вывод шестнадцатеричной цифры
;           -------------------------------------------
;   Входы: BL содержит шестнадцатеричную цифру
;   Вызовы:     Write_char
Write_digit_hex:
        .   .   .   .   .   .   .   .      ; Тело процедуры Write_digit_hex
                ret
;                   Вывод символа на экран
;                   ------------------------------
;   Входы:  DL  содержит  код символа,  выводимый  на  экран
Write_char:
                push  ax
                mov   ah,2                 ; Функция  вывода  символа
                int     21h                ; Вывод  символа  на  экран
                pop   ax
                ret
```

Лабораторная работа 6

Н а п и ш и т е на ассемблере и поместите в файл *Video_io.asm* программу вывода двузначного шестнадцатеричного числа. Все процедуры исходной программы должны содержать вводные и текущие комментарии.

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 п. 6 для проверки программы, выполняющей те же функции.

Обязательно сохраните файл *Video_io.asm*. На последующих работах 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_byte_hex* так, чтобы из нее вызывалась не процедура *Write_byte_hex*, а процедура *Write_digit_hex*. После того, как *Write_digit_hex* будет отлажена, *Test_write_byte_hex* восстанавливается, и программа отлаживается целиком.

2.3 ВЫВОД НА ЭКРАН ДЕСЯТИЧНЫХ И ШЕСТНАДЦАТЕРИЧНЫХ ЧИСЕЛ

В процессе разработки практически важных процедур, выполняющих вывод на экран десятичных и шестнадцатеричных чисел, производится изучение нескольких новых исполнительных операторов ассемблера, расширяются навыки работы со стеком, а также производится знакомство с представлением исходной программы в виде нескольких файлов.

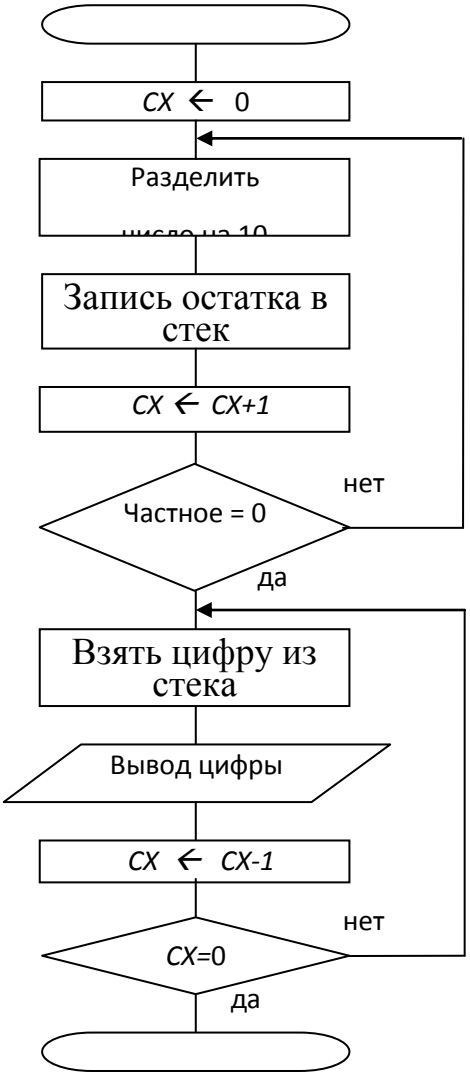
Получение алгоритма

Назовем процедуру, выполняющую вывод на экран десятичного представления слова данных, содержащего двоичное число без знака, как *Write_word_dec*. Пусть это слово данных передается на вход процедуры в регистре *DX*. Обсудим алгоритм процедуры *Write_word_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. 40.

Дерево подпрограмм

Кроме самой процедуры *Write_word_dec*, выполняющей вывод на экран десятичного представления слова данных, наша программа включает и другие процедуры (рис.41). Во-первых, это главная подпрограмма *Test_write_word_dec*, используемая для тестирования *Write_word_dec*. Во-вторых, это процедура вывода шестнадцатеричной цифры *Write_digit_hex* (она пригодна и для вывода десятичной цифры), а также процедура вывода символа *Write_char*.



CX - счетчик десятичных цифр в числе

Рис. 40**.** Алгоритм вывода десятичного числа

Главная подпрограмма *Test_write_word_dec* тестирует процедуру *Write_word_dec* с помощью числа *12345* (которое транслятор переводит в слово *3039h*):

Test_write_word_dec:

```
mov   dx,12345
call          Write_word_dec
int   20h                    ; Возврат в DOS
```

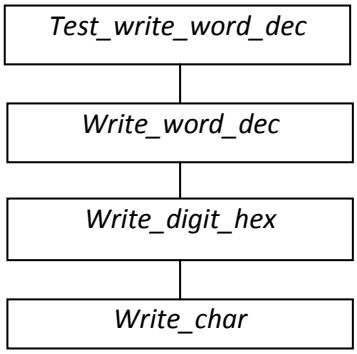


Рис. 41. Дерево подпрограмм для программы вывода десятичного числа

Запись на ассемблере

При кодировании процедуры *Write_word_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 команда ***xor*** – "исключающее *ИЛИ*". Например, команда "*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 команды "*cmp ax*,0" можно записать команду "*or ax,ax"* , используя далее или команду условного перехода ***jne*** (от "*jump if not equal*" – перейти, если не равно) или ***jnz*** (перейти, если не нуль).

Логическая команда ***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 команда 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
```

Команда *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 на предыдущей работе исходный файл *Video_io.asm*. Во-первых, с помощью текстового редактора удалите процедуру *Test_write_byte_hex* и на ее место запишите тестовую процедуру *Test_write_word_dec*.

Во-вторых, получите текст на ассемблере процедуры *Write_word_dec*, выполняющей вывод на экран содержимого 2-х байтового регистра *DX* в виде десятичного числа. При этом для кодирования этапов алгоритма "*CX* ← 0" и "Частное = 0" следует использовать описанные выше два приема. Процедуру *Write_word_dec* добавьте в конец файла *Video_io.asm*.

Внеся изменения, *п р о д е л а й т е* с *Video_io.asm* 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_word_dec* работает с беззнаковыми числами.

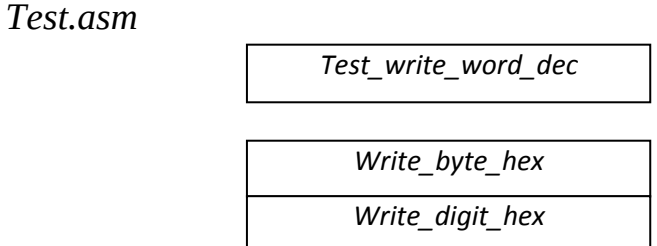
Много файловая исходная программа

Для выполнения тестирования процедуры *Write_word_dec* нам пришлось одну тестовую процедуру в файле *Video_io.asm* заменить другой, 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 данных, которым в исходной программе присвоены символьные имена. Примеры применения переменных встретятся нам в последующих работах.

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_word_dec*. Теперь уберем ее оттуда и поместим в собственный отдельный файл *Test.asm*. В результате получим файловую структуру программы, приведенную на рис. 42.



Video_io.asm

Рис. 42. Файловая структура исходной программы

Для того чтобы транслятор-ассемблер объединил несколько исходных файлов в единый текст исходной программы, каждый «родительский» исходный файл должен содержать псевдооператоры ***%include*** («включить»), в качестве операндов которых записаны имена «дочерних» файлов, заключенные в кавычки. Встретив, оператор *include*, транслятор заменяет его на текст «дочернего» исходного файла, указанного в этом операторе.

Рассмотрим эти псевдооператоры на примере файла *Test.asm*:

```
        [org   l00h]
Test_write_word_dec:
        mov    dx,12345
        call   Write_word_dec
        int    20h                    ;  Возврат в DOS
        %include    'Video_io.asm'    ; Включение файла Video_io.asm
```

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_word_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.

Лабораторная работа 7

Требуется написать и отладить программу вывода на экран шестнадцатеричных чисел *Write_word_hex* , использующую тот же алгоритм, что и приведенная выше программа вывода десятичных чисел. Тестирующая

программа и процедура *Write_word_hex* должны находиться в разных исходных файлах.

2.3 ДАМПИРОВАНИЕ ПАМЯТИ

Перейдем к разработке более сложной программы на ассемблере, выполняющей дампирование (вывод на экран) содержимого памяти подобно тому, как это делает *Debug* при выполнении команды *D*. В процессе этой разработки производится изучение новых приемов программирования на ассемблере, к которым относятся применение относительной регистровой адресации для обработки последовательности однотипных величин, а также первоначальное знакомство с системными программами *BIOS*.

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*:

```
                [org  l00h]
                jmp   Disp_line
;       Данные программы
Sector          db    10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h      ; Образец
                db    18h, 19h, 1Ah, 1Bh,1Ch,1Dh,1Eh,1Fh          ;  текста
; ---------------------------------------------------------------------------
;       Процедура дампирует 16 байт памяти в одну строку
;                       шестнадцатеричных чисел
;       ---------------------------------------------------------------------
; Вызовы:   Write_hex,  Write_char
```

```
; Чтение:     Sector
Disp_line:
                xor   bx,bx             ; Обнуление BX
                mov   cx,16             ; Счетчик байтов
.M:             mov   dl, [Sector+bx]   ; Получить один байт
                call  Write_byte_hex    ; Вывод шестн. числа
                mov   DL, ' '           ; Вывод на экран
                call  Write_char        ;  пробела
                inc   bx                ; Возврат за следующим
                loop  .M                ;  байтом
                int   20h               ; Возврат в DOS
                %include 'Video_io.asm'         ;       Подсоединение
процедур
```

В начале программы находится оператор безусловного перехода, осуществляющий переход через область данных, расположенную после оператора перехода, на начало области исполнительных операторов программы. Для исполнительных операторов новым является использование относительной регистровой адресации в операторе:

```
Hex_loop:   mov   dl, [Sector+bx]
```

Байт программы, отмеченный меткой *Sector*, имеет смещение *259 (100h+3)* относительно самого первого байта сегмента памяти, выделенного программе (команда *jmp* имеет длину три байта). Тогда при выполнении на ЦП машинной команды *mov*, соответствующей записанному оператору, реальный адрес второго операнда будет определен по формуле:

$$R = (DS) * 16 + 259 + (BX).$$

Задав нулевое содержимое регистра *BX*: (*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*, а затем получите 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, дальнейшей целью наших лабораторных работ является создание программы, позволяющей выполнять редактирование 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 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:

```
%define     cr     13              ; Возврат  каретки
```

```
        %define     lf      10          ; Перевод строки
;               Перевод строки экрана
;               --------------------------------
Send_crlf:
        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
```

Псевдооператор *%define* указывает транслятору, что имя *cr* эквивалентно числу 13, а имя *lf* эквивалентно числу 10. Поэтому везде в программе, где транслятор встретит данное имя, он заменит его указанным числом. Применение данного псевдооператора позволяет программисту записывать в программе вместо чисел более удобные их символьные обозначения. Например, вместо 13 записывать *cr*.

На рис. 43 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.

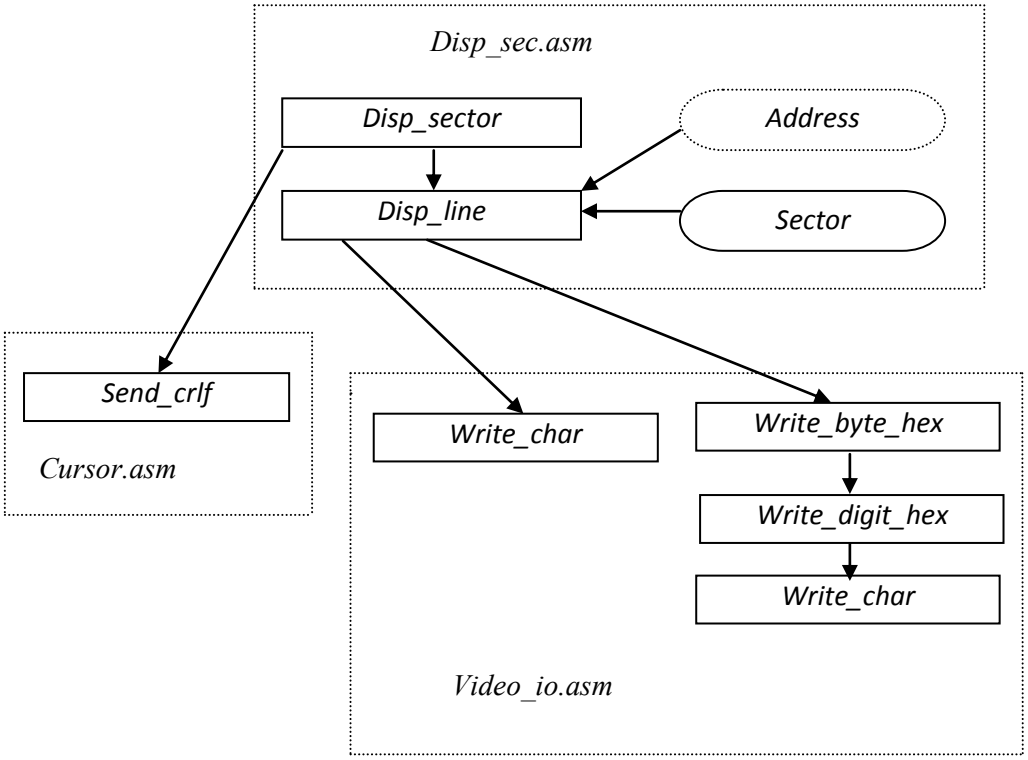


Рис. 43. Дерево подпрограмм и файловая структура программы вывода на экран сектора

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

```
            [org    l00h]
            jmp    Disp_sector
;     Данные программы
Sector    times  16     db  10h          ;  Первая строка байтов
            times  16     db  11h
          . . . . . . . . . . . . . . . . . . . .
            times  16     db  1Fh          ;  Последняя  строка  байтов
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;              Отображает  на  экран  сектор (256 байт)
;              ----------------------------------------------------
;   Вызовы:    Disp_line,  Send_crlf
Disp_sector:
            xor          dx,dx              ;  Начало  Sector
            mov          cx,16              ;  Число строк 16
```

```
.M:     call    Disp_line       ; Вывод строки
        call    Send_crlf       ; Перевод строки
        add     dx,16           ; Номер следующего байта
        loop    .M              ; Проверка числа строк
        int     20h             ; Выход в DOS
; 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:
        push  bx
        push  cx
        push  dx
        mov   bx,dx             ; В  BX номер первого байта
        mov   cx, 16            ; Счетчик байтов
.M:     mov   dl, Sector[bx]    ; Получить один байт
        call  Write_byte_hex    ; Вывод шестнад. числа
        mov   dl, ' '           ; Вывод  на  экран
        call  Write_char        ;   пробела
        inc   bx                ; Возврат  за
        loop  .M                ;   следующим байтом
        pop   dx
        pop   cx
        pop   bx
        ret
        %include    'Video_io.asm'    ; Подсоединение процедур
        %include    'Cursor.asm'      ;              --//--
```

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 файла *Disp_sec*. 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 *BIOS*.

Операционная система, например *DOS*, использует подпрограммы *BIOS* для выполнения своих операций обмена с внешними устройствами. Разработчик прикладной программы может применять подпрограммы *BIOS* точно так же, как и подпрограммы *DOS*, т.е. с помощью команд *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 оператором "*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:

```
;               Очистка экрана
;               --------------------
Clear_screen:
        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
```

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* оператор вызова процедуры очистки экрана.

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*, которая восстанавливает на экране свою информацию. Для избежания этого в конце процедуры *Disp_sector* поместите оператор, выполняющий ввод символа с клавиатуры. Наличие такого оператора позволит наблюдать на экране неискаженное изображение сектора до тех пор, пока не будет нажата любая клавиша. Впоследствии у нас отпадет потребность в подобном "торможении" нашей программы.

Лабораторная работа 8

Требуется скорректировать 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-слово *Address* содержит начальный адрес-смещение 256-и байтовой области (рис.43). Поэтому начальный адрес строки рассчитывается в процедуре *Disp_line* путем суммирования содержимого *Address* с содержимым регистра *DX*. Это суммирование выполняет оператор:

```
add     dx, [Address]
```

Обратите внимание, что имя переменной *Address* заключено в квадратные скобки. При отсутствии этих скобок к содержимому регистра *dx* было бы прибавлено не содержимое переменной, а ее адрес-смещение.

Не забудьте определить переменную-слово *Address* в начале файла *Disp_sec.asm*, задав ей первоначальное значение, например 100*h*. Для этого

используется псевдооператор резервирования слова памяти ***dw***:

Address dw 100h

В последующих работах будут созданы процедуры, которые загружают в переменную *Sector* редактируемый фрагмент ОП, а в переменную *Address* – начальный адрес (внутрисегментное смещение) этого фрагмента.

Примечание 2. Перед оператором с меткой "*.M*" 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"*).

2.4 ПЕРЕПИСКА СЕКТОРА ПАМЯТИ

Разработанная в предыдущем разделе программа выполняет вывод на экран единственного 256-байтового сектора, заполняемого транслятором до начала выполнения программы. Теперь пришла пора заняться выводом на экран любого сектора, входящего в состав того сегмента ОП, который выделен нашей программе. Напомним, что размер одного сегмента ОП – 64*K*, что соответствует 256 сегментам по 256 байт.

Функции переписки сектора

В полученной ранее программе мы использовали для хранения редактируемого сектора (256 байт) область (переменную) *Sector*. В принципе можно было бы не использовать эту область, а напрямую работать с требуемым сектором памяти. Но в этом случае любая ошибка в редактировании мгновенно отразилась бы на оригинале. Поэтому мы будем копировать текущий сектор памяти в область *Sector*, а после выполнения редактирования осуществим обратное копирование.

На рис.44 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:

Write_sector – переписывает содержимое переменной *Sector* в

область памяти, начальный адрес которой находится в переменной *Address*;

Init_sector помещает в переменную *Sector* начальный сектор сегмента памяти нашей программы. Кроме того, эта процедура помещает *0* в переменную *Address*;

Prev_sector помещает в *Sector* предыдущий сектор памяти, а в переменную *Address* записывает начальный адрес этого сектора;

Next_sector помещает в переменные *Sector* и *Address* соответственно содержимое и начальный адрес следующего сектора памяти;

N_sector загружает в область *Sector N*-й сектор сегмента программы. При этом значение *N* ($0 \le N \le 255$), умноженное на 256, записывается в переменную *Address*.

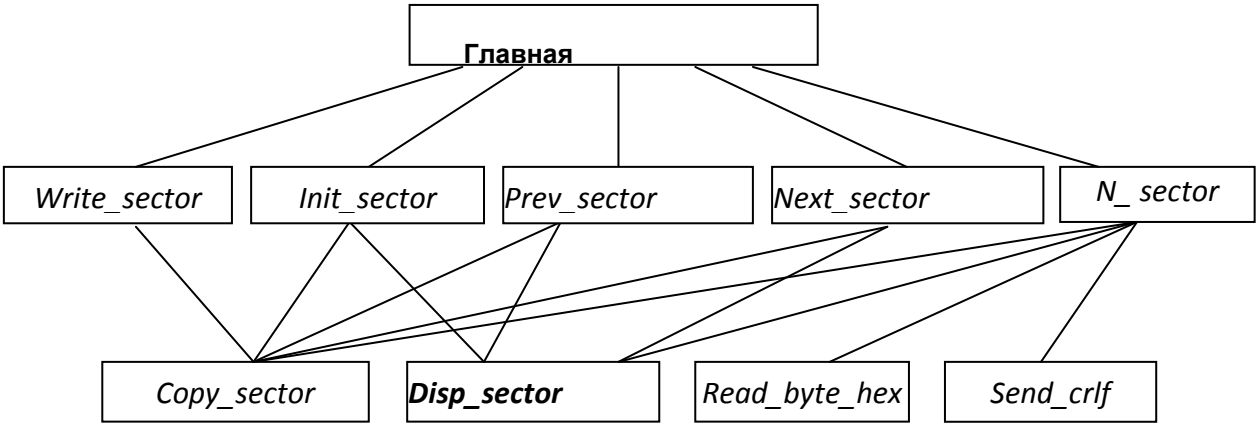


Рис. 44. Дерево подпрограмм для программы, выполняющей функции переписки сектора

Копирование сектора

Как видно из рис.44, все процедуры, выполняющие функции по переписке сектора, пользуются услугами процедуры *Copy_sector*, которая записывает содержимое заданного сектора памяти в качестве содержимого другого заданного сектора. *Copy_sector* имеет два входных параметра: регистр *SI* содержит адрес (внутрисегментное смещение) сектора-источника, *DI* – адрес-смещение сектора-приемника.

При написании процедуры *Copy_sector* мы могли бы ограничиться уже известными нам операторами, но мы привлечем для этого ряд новых операторов:

```
;                Копирует  сектор (256 байт)
;                ---------------------------------------
;     Входы:   SI  – начальный адрес сектора-источника
;                DI – начальный адрес сектора-приемника
Copy_sector:
        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
```

Строковый (цепочечный) оператор ***movsb*** занимает центральное место в *Copy_sector*. Взятый без префикса ***rep***, он 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 специальный оператор ***cld***, а установку – оператор ***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 операторы ***pushf*** и ***popf***, выполняющие запись в стек и извлечение оттуда регистра флагов.

Префикс повторения команды ***rep*** многократно усиливает мощность строковой команды. При этом оператор *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 следующий за *movsb* оператор.

Перед вызовом процедуры *Copy_sector* необходимо в вызывающей ее программе загрузить в регистры *SI* и *DI* начальные адреса секторов памяти. Для этого можно использовать оператор *mov*. Например, в результате оператора "*mov si,Sector"* адрес самого первого байта области *Sector* будет помещен в регистр *SI*.

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

Алгоритмы процедур

На рис.45 приведён алгоритм процедуры *Init_sector*, на рис.46 алгоритм *Prev_sector*, а на рис.47 алгоритм *N_sector*. Что касается алгоритма процедуры *Next_sector*, то он очень похож на алгоритм *Prev_sector*. Отличие состоит в том, что номер текущего сектора *N* сравнивается не с 0, а с числом *FFh*.

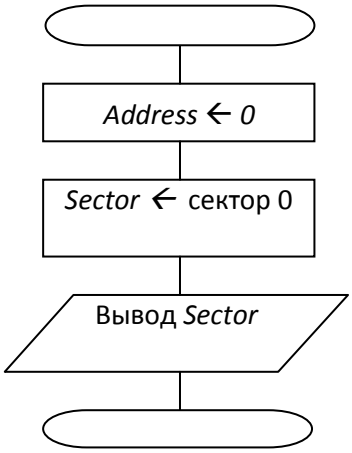
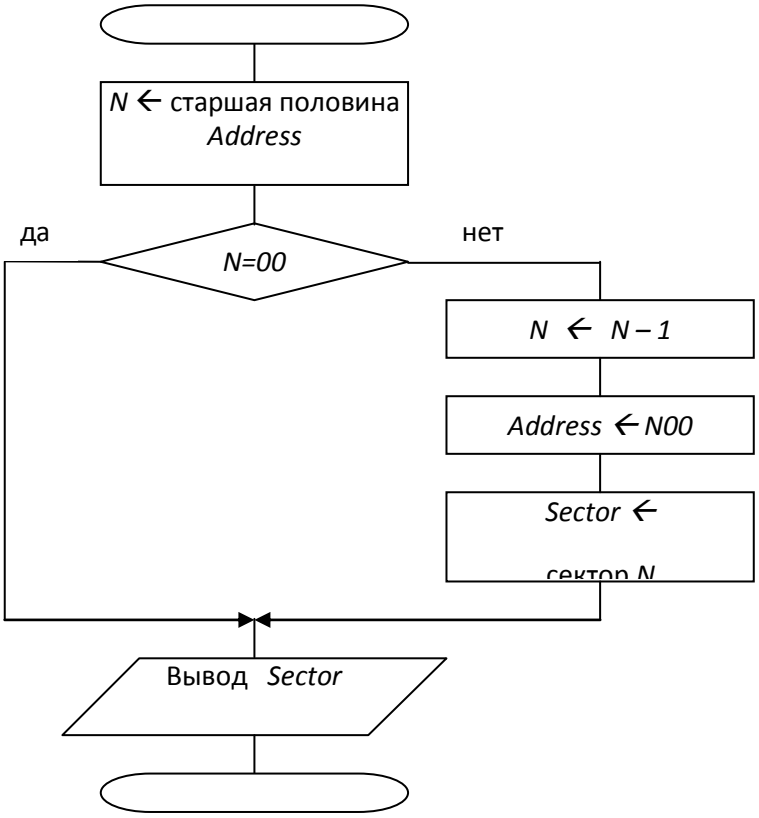


Рис. 45. Алгоритм процедуры *Init_sector*

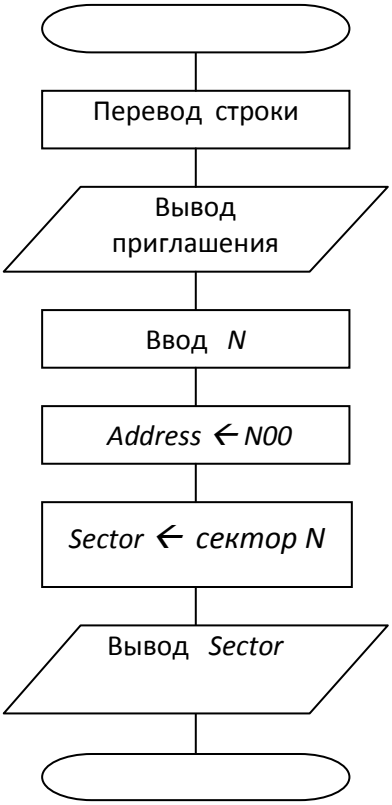
З а п и ш и т е тексты процедур *Write_sector*, *Init_sector*, *Prev_sector*, *Next_sector*, *N_sector* в файл *Disp_sec.asm*.

Примечание. Реализация этапа "Ввод *N*" в процедуре *N_sector* осуществляется путём вызова процедуры *Read_byte_hex*, выполняющей ввод с клавиатуры 2-х значного шестнадцатеричного числа. *Read_byte_hex*, в свою очередь, вызывает процедуру *Read_digit_hex*, выполняющую ввод шестнадцатеричной цифры. Алгоритмы обеих процедур были рассмотрены нами ранее в п.8.3. *З а п и ш и т е* тексты этих процедур в новый файл *Kbd_io.asm*.



N-номер текущего сектора

Рис. 46. Алгоритм процедуры *Prev_sector*



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

Рис. 47. Алгоритм процедуры *N_sector*

Лабораторная работа 9

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:
        call    Init_sector
        mov     ah,1            ; Ожидание нажатия
        int     21h             ;  клавиши
        call    N_sector
        mov     ah,1            ;       --//--
        int     21h
        call    Prev_sector
        mov     ah,1            ;       --//--
        int     21h
        call    Next_sector
        mov     ah,1            ;       --//--
        int     21h
        int     20h
```

Данную программу следует выполнить несколько раз, задавая при выполнении процедуры *N_sector* различные номера секторов. Это позволяет сделать текущим любой интересующий нас сектор. Особое внимание следует уделить граничным условиям – первому и последнему секторам.

2.5 ДИСПЕТЧЕР КОМАНД

В процессе выполнения работы решается задача разработки диспетчера – программного модуля, обеспечивающего диалог с пользователем и выполняющего координацию работы других модулей программы (текстового редактора). Применяемый при этом подход может быть использован для построения диспетчеров в других программных системах.

Ввод команд

Почти любая полноценная программа является интерактивной, т.е. способной вести диалог со своим пользователем. Естественно, что наш редактор текстовой информации также должен быть интерактивным. Он должен 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.

Допустим, что наш редактор 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 редактора.

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 команды *”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 команду *”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.

```
;          Считывает  код  символа  с  клавиатуры
;          - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;   Выход:  AL – код ASCII символа
;              AH – тип символа (1 – обычный символ;  -1 – символ
;                        расширенного кода)
Read_byte:
                    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
```

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 команды *ret*. Нажимая различные клавиши, проверьте правильность заполнения процедурой регистра *AX*.

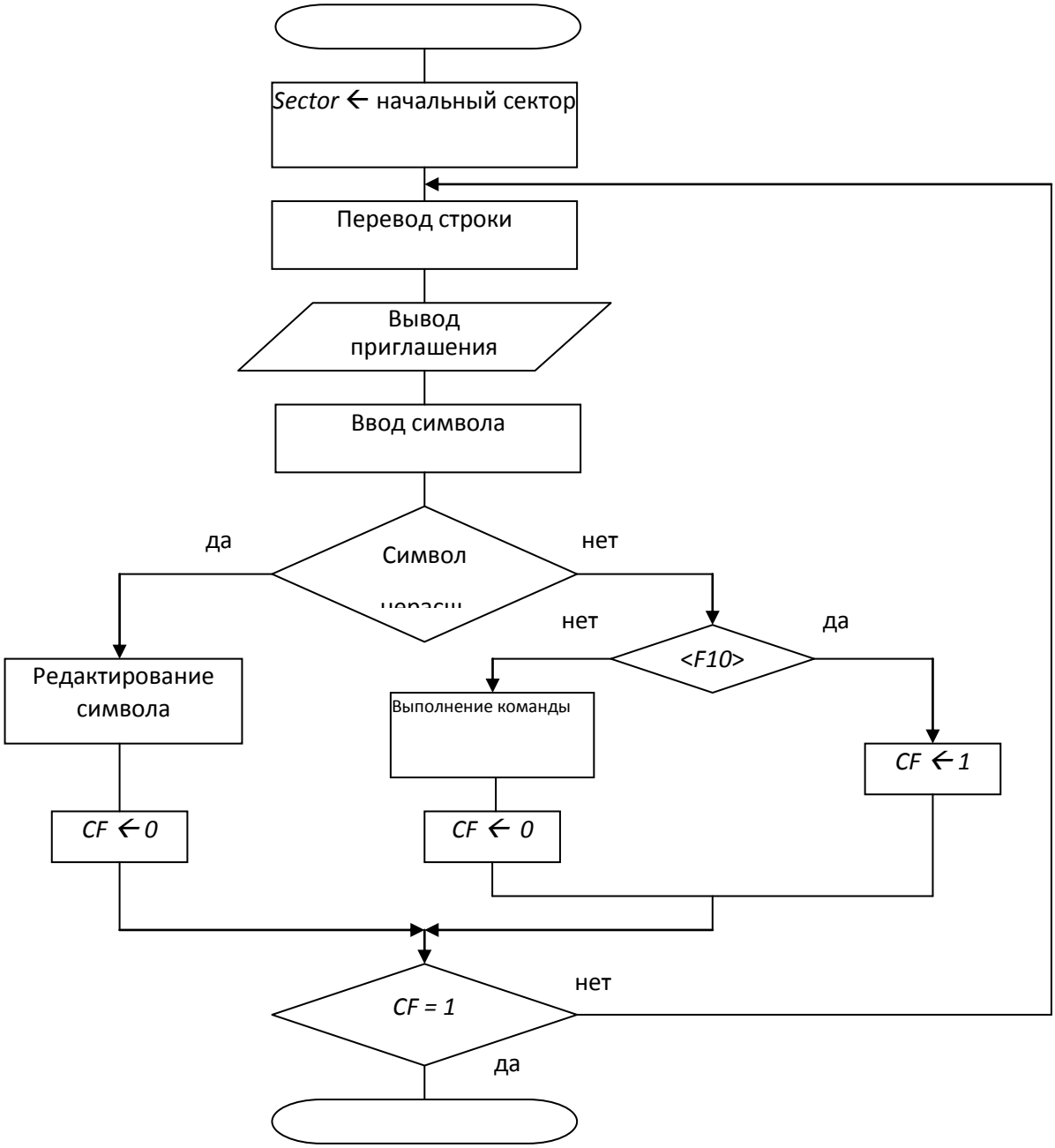
Алгоритм диспетчера

На рис. 48 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 редактор ожидает команд пользователя. Два этапа – «Р 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 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*.



Sector – буфер для редактирования сектора

CF – признак завершения (*0* – продолжить, *1* – окончить работу)

Рис.48. Алгоритм процедуры *Dispatcher*

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.

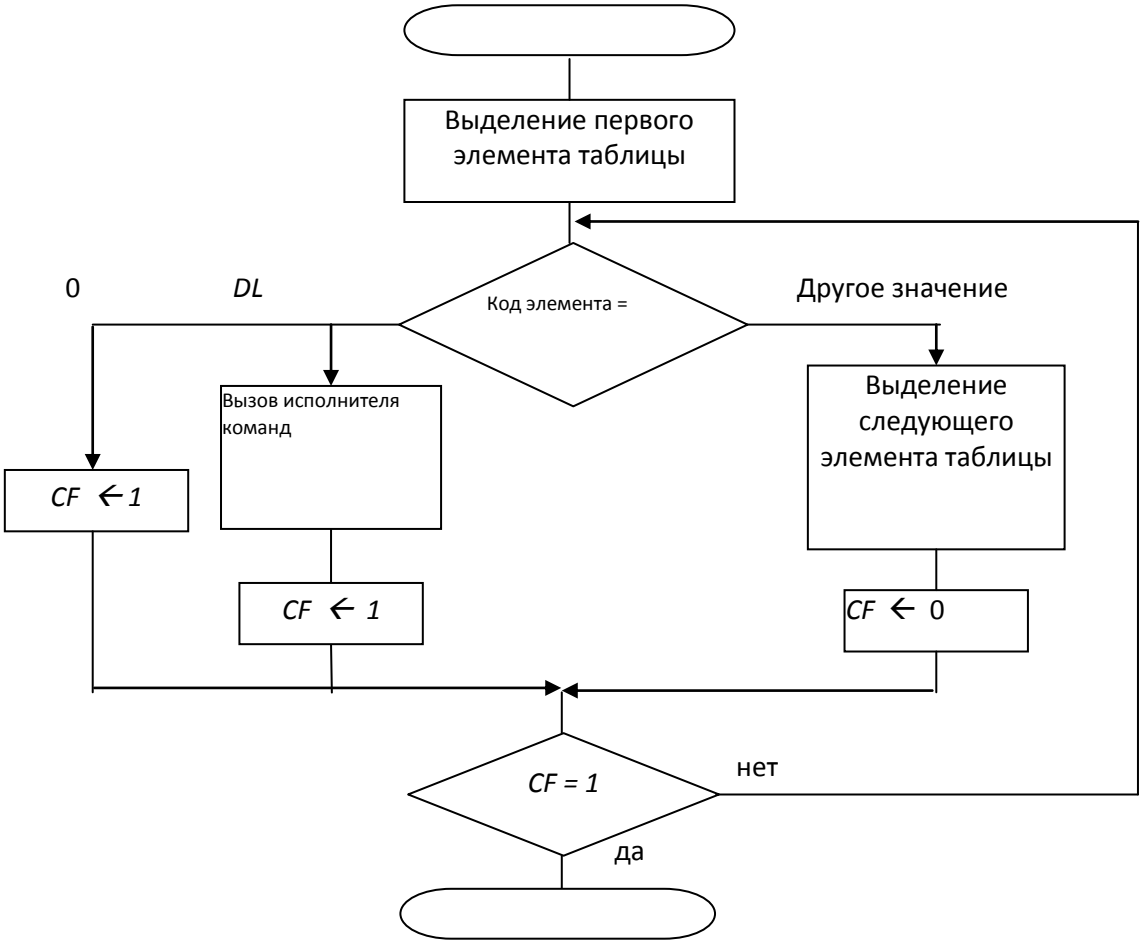
На рис. 49 приведена блок-схема процедуры *Command*. Центральной

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*.

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*. Заключительный фрагмент этого файла, содержащий модули *Command* и *Table*:

```
                [org  100h]
;       Координирует выполнение модулей редактора
;       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Dispatcher:
    .   .   .   .   .   .   .   .   .     ; Тело процедуры Dispatcher
                ret
;           Интерпретатор команд
;           - - - - - - - - - - - - - - - -
;    Входы:   DL - код ASCII, соответствующий команде
;    Чтение:  Table – таблица переходов
Command:
                push  bx
                mov   bx,Table              ; BX←адрес таблицы
M1:             cmp   BYTE  PTR [BX], 0     ; Конец таблицы ?
                je    Exit                  ; 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  [bx]                ; Вызов процедуры
Exit:       stc                       ;  CF ← 1
M2:         jnc   M1    ;Повторение для нового элемента таблицы
            pop   bx
            ret
;     Таблица содержит разрешенные расширенные ASCII-коды и
;      адреса процедур,  выполняющих соответствующие команды
Table       db    3bh                       ; <F1>
            dw    Prev_sector
            db    3ch                       ; <F2>
            dw    Copy_sector
            db    3dh                       ; <F3>
            dw    Next_sector
            db    3eh                       ; <F4>
            dw    Init_sector
            db    3fh                       ; <F5>
            dw    N_sector
            db    0                         ; Конец таблицы
```



DL – код команды

CF – признак завершения (*0* – продолжить, *1* – завершить работу)

Рис. 49. Алгоритм процедуры *Command*

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*.

Лабораторная работа 10

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.