ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА
(РОСАВИАЦИЯ)

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ» (МГТУ ГА)

Кафедра вычислительных машин, комплексов, систем и сетей

Н.И. Черкасова

ПРОГРАММИРОВАНИЕ НА МАШИННО-ОРИЕНТИРОВАННОМ ЯЗЫКЕ

Учебно-методическое пособие
по выполнению лабораторной работы № 1

для студентов II курса
направления 09.03.01
очной формы обучения

Москва
ИД Академии Жуковского
2021

УДК 004.43
ББК 6Ф7.3
Ч-48

Рецензент:
Надейкина Л.А. – канд. физ.-мат. наук, доцент

Черкасова Н.И.

Ч-48 Программирование на машинно-ориентированном языке [Текст] : учебно-методическое пособие по выполнению лабораторной работы № 1 / Н.И. Черкасова. – М.: ИД Академии Жуковского, 2021. – 32 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Программирование на машинно-ориентированном языке» по учебному плану для студентов II курса направления 09.03.01 очной формы обучения.

Рассмотрено и одобрено на заседаниях кафедры 25.05.2021 г. и методического совета 25.05.2021 г.

УДК 004.43
ББК 6Ф7.3

В авторской редакции

Подписано в печать 25.10.2021 г.
Формат 60x84/16 Печ. л. 2 Усл. печ. л. 1,86
Заказ № 796/0616-УМП09 Тираж 30 экз.

Московский государственный технический университет ГА
125993, Москва, Кронштадтский бульвар, д. 20

Издательский дом Академии имени Н. Е. Жуковского
125167, Москва, 8-го Марта 4-я ул., д. 6А
Тел.: (495) 973-45-68
E-mail: zakaz@itsbook.ru

Содержание

ЛАБОРАТОРНАЯ РАБОТА№1
Адресация и ввод/вывод в программе на Ассемблере

1. Цель лабораторной работы

Целью лабораторной работы является освоение:

1. Директив определения данных
2. Способов адресации операндов
3. Особенностей ввода/вывода в программе на Ассемблере

2.Содержание отчёта

Отчет по лабораторной работе должен включать:
1) цель лабораторной работы;
2) конкретный вариант задания на выполнение;
3) тексты программ;
4) схемы алгоритмов;
5) результаты выполнения программ.

3.Краткие теоретические сведения

3.1. Операнды и типы команд ассемблера

Под операндами понимаются данные, обрабатываемые командой ассемблера. Каждый операнд так или иначе задается командой ассемблера. Количество операндов в машинных командах может изменяться от 0 до 2-х. Большинство команд оперирует с двумя операндами. В командах различают операнды источники информации и операнды приемники (операнды назначения) информации. В некоторых случаях операнды могут задавать как источник, так и приемник данных. В некоторых машинных командах операнд явно не задается, а подразумевается. Эта особенность команд затрудняет как написание программ на ассемблере, так и их интерпретацию.

Команды, исполняемые процессором, обрабатывают информацию, которая может находиться, например, на регистре процессора или в оперативной памяти. Тогда команда процессора обрабатывает содержимое регистра или содержимое оперативной памяти соответственно. Таким образом, операндом команды может быть содержимое регистра или содержимое оперативной памяти.

Рассмотрим возможные сочетания операндов в командах ассемблера. Термином память обозначается оперативная память компьютера, так как регистры - это тоже память, но ее более точное название - «регистровая память» [1].

Один из операндов должен находиться в регистре или задаваться непосредственно в команде, то есть быть непосредственным операндом.
Однако, имеются исключения из правил сочетания операндов в командах. Такие исключения возникают в следующих случаях:

- в командах работы с цепочками, которые могут перемещать данные из памяти в память;
- в командах работы со стеком, которые могут переносить данные между стеком и памятью, а стек – это область памяти;
- в командах типа умножения (к ним относятся и команды деления), которые кроме операнда, указанного в команде, используют еще и другие операнды, не указанные в команде явно.

Операндами могут быть:

- константы, например, - числа;
- содержимое регистра;
- содержимое ячейки памяти;
- значения переменных, заданных символьными идентификаторами;
- выражения: комбинации чисел, имен регистров, ячеек памяти, идентификаторов с арифметическими, логическими, побитовыми и атрибутивными операторами.

Рассмотрим общие характеристики команд процессора:

• Процессор может выполнять команды только из оперативной памяти. Адрес текущей команды: cs:eip.
• Команда может иметь префиксы, влияющие на ее выполнение, например, - префикс повторения.
• Как правило, недопустимо использовать более одного операнда "память" в команде. При работе с несколькими ячейками используют промежуточные регистры или стек.
• Обычно, размер операндов должен совпадать.
• Как правило, команды, имеющие два операнда, производят действие над первым операндом (приемником) при помощи второго (источника).
• Обычно арифметические и логические команды устанавливают флаги в соответствии с результатом.
• Команда может вызвать исключение. Если исключение не предусмотрено, появится сообщение об ошибке программы.
• Некоторые команды имеют несколько названий. Например, nop и xchg eax, eax – одна и та же команда.
• В большинстве случаев, командам, работающим с операндом в слово, соответствует машинный код, аналогичный команде, работающей с двойным словом. Перед командой компилятор ставит префикс замены разрядности, если она не совпадает с текущей. Лучше не использовать команды, не совпадающие с текущей разрядностью, без особой необходимости. В программах под ОС Windows желательно использовать байты или двойные слова в качестве операндов[2].

• Многие команды имеют короткую форму. Если операнд (знаковое число) помещается в байт, команды обычно занимают меньше места. Например, команда mov eax,10 занимает 5 байт, а две команды push 10; pop eax только 3.

Далее приведено описание наиболее часто используемых команд в прикладных программах.

Таблица 1 Логические команды, работа с битами:

Команда	Количество операндов	Назначение команды
and or xor	2	Операнд1 = операнд1 and / or / xor операнд2
bt btc btr bts	2	Установить флаг CF значением бита номер операнд2 из операнда1. c – изменить его значение на обратное, r – сбросить, s – установить
not	1	изменить значение всех бит операнда на обратное
shl shr	2	сдвиг битов операнда 1 влево / вправо на операнд2 (число или cl). CF устанавливается значением выдвигаемого бита

Пример:
and eax,100011b сброс всех бит регистра кроме нулевого, первого и пятого
or eax,111b установить младшие 3 бита регистра eax
or ecx,ecx Используется для установки флагов. Занимает 2 байта вместо cmp ecx,0 – три
xor dword ptr [AnyVal],0F0h инверсия бит 4,5,5,7 переменной AnyVal
xor eax,eax обнулить значение регистра eax. Команда занимает 2 байта вместо mov eax,0 – пять
Прочие команды приведены в таблице 2.

Таблица 2. Команды Ассемблера

Команда	Количество операндов	Назначение команды
cdq	0	установить все биты регистра edx значением старшего бита регистра eax. Обычно используется перед командой div
stc clc	0	Установить / сбросить флаг CF. Иногда используется для возврата результата выполнения процедуры

cmp	2	Сравнить операнды. Установить флаги как при команде sub
test	2	Установить флаги, как при команде and
lea	2	вычислить выражение типа «Адресация Памяти», и поместить в первый операнд. Используется для простых вычислений
bswap	1	поменять местами байты #0, 3 и #1,2 операнда
mov	2	Поместить значение второго операнда в первый
movsx movzx	2	поместить в первый операнд (word или dword) значение второго (word или byte). Остальные биты приемника заполнить: знаковым битом источника / нулями.
Xchg	2	поменять значения операндов
nop	0	команда, не выполняющая действий, занимает 1 байт

Таблица 3. Команды пересылки

Команда	Количество операндов	Назначение команды
mov	2	Поместить значение второго операнда в первый
movsx movzx	2	поместить в первый операнд (word или dword) значение второго (word или byte). Остальные биты приемника заполнить: знаковым битом источника / нулями.
Xchg	2	поменять значения операндов

Таблица 4. Команды работы со стеком:

Команда	Количество операндов	Назначение команды
pop	1	извлечь число из стека в операнд
popa	0	извлечь регистры общего назначения после команды pusha (кроме esp)
popf	0	извлечь число из стека в регистр eflags
push	1	поместить операнд в стек
pusha	0	поместить регистры eax ecx edx ebx esp ebp esi edi в стек. Регистр edi оказывается на вершине

pushf	0	поместить eflags в стек

3.2. Директивы определения данных

Директивы определения данных (или директивы резервирования и инициализации данных) необходимы для описания типов переменных (байт, слово, двойное слово и т. д.), с которыми работает программа. Директивы определения данных являются указаниями транслятору на выделение определённого объёма памяти. Директива начинается с D (Define, определить), после которой идет сокращение от размера определяемого элемента данных (DB — Byte байт, DW — Word слово, DD — Double word двойное слово, DF — Far pointer word указатель на дальнее слово, DP — Pointer указатель, DQ — Quadword учетверенное слово, DT — Ten bytes 10 байт). DF — длинный сегментированный указатель (32 бита смещения + 16 бит сегмента), синоним: DP.

Кроме целых чисел, как аргументы, можно указывать вещественные. DD может также использоваться для описания и хранения коротки@ (singleprecision) вещественны@ чисел +/-1,3x*10-3x …3,40*103x, DQ для длинны@ (long, double) вещественны@ +/-2,33*10-30x... 1,79*1030x, а DT для временны@ (temporary) вещественны@ +/-3,37*10-4932... 1,1x*104932 (табл.5, табл.6).

Помимо интерпретации значений в соответствующих ячейках как целых, они могут интерпретироваться следующим образом: DW — int/unsigned или 16-битное смещение; DD — long, float, 16-битный far (сегмент:смещение) или 32-битный близкий (смещение) указатель; DF — 32-битный far (дальний) указатель типа сегмент:смещение; DQ — double; DT — упакованное десятичное (packed BCD) число длинной 20 цифр или long double (оба — форматы представления числа для сопроцессора). Все, выше перечисленные директивы можно использовать для строкового значения (представление чисел в виде ASCII кодов)[3].

Пример:

String_1 DB 'a'

String_2 DW 'ba'

String_3 DD 'dcba'

String_4 DB 'я программирую на ассемблере!' Директива DB резервирует и инициализирует столько байт, сколько символов написано между кавычками или апострофами.

Таблица 5. Директивы определения данных

Директива и ее синонимы	Размер в		в десятеричной

	байта F	в шестнадцатеричной системе счисления	беззнаковые целые числа
DB/BYTE/SBYTE	1	От 0 до 0FFh	От 0 до 255 O
DW/WORD/SWORD	2	От 0 до 0FFFFh	От 0 до 65535
DD/DWORD/SDWORD/REAL4	4	От 0 до 0FFFFFFFFh	От 0 до 232-1
DF/ DP/FWORD	6	От 0 до 0FFFFFFFFFFFh	От 0 до 248-1
DQ/QWORD/REAL8	8	От 0 до 0FFFFFFFFFFFFFFFh	От 0 до 264-1
DT/TBYTE/REAL10	10	От 0 до 0FFFFFFFFFFFFFFFFFFh	От 0 до 280-1

Таблица 6. - Интерпретация директив DW, DD, DP и DQ как адресные выражения

Директива	Адресное выражение
DW	16-битный адрес сегмента; смещение в 16-битном сегменте
DD	16-битный адрес + 16-битное смещение; 32-битный адрес
DF	16-битный адрес сегмента + 32-битное смещение
DQ	16-битный адрес сегмента + 64-битное смещение

Директива DB

По директиве DB (Define Byte, определить байт) определяются данные размером в байт

Синтаксис директивы:
[<*имя*>] DB <*операнд*> {,<*операнд*>} [;<*комментарий*>]
Синонимы: BYTE (от 0 до 255), SBYTE (от –128 до +127)
Встретив такую директиву, ассемблер вычисляет операнды и записывает их значения в последовательные байты памяти. Первому из этих байтов дается указанное имя, по которому на этот байт можно ссылаться из других мест программы.
Существуют следующие способы задания операндов директивы DB:

1. ? (знак неопределенного значения);
2. константное выражение со значением от –128 до 255;

3. символьная строка из одного или более символов, заключенная в кавычки.

Операнд «?»

Пример: X DB ?

По этой директиве отводится один байт в памяти, в который ничего не записывается, метке X присваивается адрес этого байта. В этом случае переменная Q не получила никакого значения. Выделив байт под переменную, ассемблер запоминает ее адрес. Когда ассемблер снова встретит в тексте программы имя этой переменной, то он заменит имя на данный адрес.

Адрес ячейки, выделенной переменной с именем X, принято называть значением имени X. По описанию переменной ассемблер запоминает, сколько байт занимает переменная в памяти. Этот размер называется типом переменной. Значение (адрес) и тип (размер) имени переменной однозначно определяют ячейку памяти, обозначаемую этим именем.

Операнд - константное выражение со значением от -128 до 255

Применяется для описания переменной размером в байт с начальным значением в виде числа величиной от –128 до 255. Один байт представляет либо знаковое число в пределах от –128 до +127, либо беззнаковое число от 0 до 255. Максимальное число 255 равно 2x – 1 (x по количеству битов в байте).

Пример:

A DB 254 ;0FEh

B DB –2 ;0FEh (256–2=254)

C DB 0FEh ;0FEh

D DB 11111110b ;0FEh

По каждой из этих директив ассемблер отводит один байт под переменную и записывает в этот байт число. К началу выполнения программы переменная A будет иметь значение 254, переменная B — значение — 2, переменная C — значение 0FEh, а переменная D 11111110b. В качестве значения переменной может быть указан символ. В качестве символа можно указать как его код (в кодировке ASCII) либо указать сам символ в кавычках.

Директива с несколькими операндами

Для описания переменной-массива с некоторыми начальными значениями применяется директива DB с несколькими операндами.

Пример:

M DB 2 DB -2 DB ?

DB ’*’

В массивах имя дается только его первому элементу, а остальные остаются безымянными. Если в массиве много элементов, то такой способ описания массива слишком громоздок. Поэтому допускается также и упрощенная форма записи:

M DB 2,-2,?,’*’

По директиве DB с несколькими операндами ассемблер выделяет в памяти соседние байты памяти по одному на каждый операнд, и записывает в эти байты значения операндов (для операнда «?» ничего не записывается).

Операнд - строка

Если в директиве несколько соседних операндов символы, то их можно объединить в одну строку. Два следующих примера эквивалентны:

S DB ’к’,’о’,’т’

S DB ’кот’

Вопрос о том, объединять соседние символы в одну строку или нет, а если объединять, то какие именно, решает сам автор программы. Правильной будет и такая запись:

S DB ’ко’,’т’

такая

S DB ’к’,’от’

В любом случае каждая из этих директив является эквивалентом следующей директивы:

S DB ’к’

DB ’о’

DB ’т’

Операнд - конструкция повторения DUP

Часто в директиве необходим указывать одинаковые операнды. Например, при описании байтового массива R из x элементов, где каждый элемент проинициализирован 0, можно записать так:

R DB 0, 0, 0, 0, 0, 0, 0, 0

А можно записать короче:

R DB x DUP (0)

В этой директиве в качестве операнда использована конструкция повторения, в которой сначала указывается коэффициент повторения, затем служебное слово DUP (**DUP**LICATE *копировать*), а за ним в круглых скобках — повторяемая величина.

В общем случае эта конструкция имеет следующий вид: *k* DUP (p_1, p_2, …, p_n), где *k* — константное выражение с положительным значением, n>=1, p_i — любой допустимый операнд директивы DB (в частности, это может быть снова конструкция повторения):

Например, директивы слева эквивалентны директивам справа:

X DB 2 DUP (’ab’, ?, 1)	X DB ’ab’,?,1,’ab’,?,1
Y DB -7,3DUP(0,2DUP(?))	Y DB -7,0,?,?,0,?,?,0,?,?

Вложенность конструкций DUP используют для описания многомерных массивов. Директива A DB 200 DUP (300 DUP (?)) отводит место в памяти под байтовую матрицу A размера 200*300, в которой элементы расположены в памяти следующим образом: первые 300 байтов — это элементы первой строки матрицы, следующие 300 байтов — элементы второй строки и т.д.

Директива DW

Директивой DW (**D**efine **W**ord, *определить слово*) описываются переменные размером в слово. Аналогична директиве DB
Синонимы: WORD (0 до 65535), SWORD (от -2^{15} до $+2^{15}-1$)
Операнд «?»
Пример: X DW ? По этой директиве описывается переменная X. Для нее отводится одно слово в памяти, в которое ничего не записывается, т.е. эта переменная не получает начального значения.
Константное выражение со значением от -32768 до 65535
Применяется для описания переменной размером в слово с начальным значением в виде числа величиной от -32768 до 65535. Слово представляет либо знаковое число в пределах от -32768 до +32767, либо беззнаковое число от 0 до 65535. Максимальное число 65535 равно $2^{16} - 1$ (16 по количеству битов составляющих слово),
например:

```
A  DW 1234h
B  DW –2 ;0FFFEh(65536–2=65534)
```

По каждой из этих директив ассемблер отводит одно слово под переменную и записывает в это слово указанное число, которое становится начальным значением этой переменной. Как и в случае директивы DB, неотрицательные числа записываются в память как числа без знака, а отрицательные числа в дополнительном коде. Поэтому числа, которые могут быть заданы как операнды директивы DW, должны принадлежать отрезку $[-2^{15}, 2^{16} - 1]$.
В памяти компьютера числа размером в слово хранятся в «перевернутом» виде, поэтому по этим двум директивам память заполнится следующим образом:
Представление в памяти директивы DW 1234h,-2 Частный случай директивы DW строка из одного или двух символов,
например:

```
S1 DW ’01’
S2 DW ’1’
```

Если указана строка из двух символов, то ассемблер берет коды указанных символов (код ASCII для ‘0’ равен 30h, для ‘1’ - 31h) и образует из них число-слово (3031h), которое и считается начальным значением описываемой переменной S_1. Но, как и любое число размером в слово, данное значение будет записано в память в «перевернутом» виде. Если же в правой части директивы DW указан один символ, то к нему слева ассемблер приписывает символ с кодом 0 и далее работает с этим символом как с двухсимвольной строкой. В связи с тем, что операторы-строки записываются в память в «перевернутом» виде, что, в общем-то, не характерно для строк, то подобные операнды редко указываются в директиве D. Первые 128 символов Unicode с кодами от 0000h до 007Fh (цифры, знаки препинания, пробел, символы от ‘A’ до ‘Z’ и от ‘a’ до ‘z’ и т.п.) совпадают с символами ASCII, поэтому для

преобразования ASCII символов в UNICODE символы используется следующая формула:
16-битный номер символа в UNICODE = = (с 15-го по x-ой биты нули) + (номер символа в ASCII)
Тогда через директиву dw можно передавать латиницу в формате UNICODE следующим образом: dw ‘H’,‘e’,‘l’,‘l’,‘o’,‘,’,‘ ’,‘w’,‘o’,‘r’,‘l’,‘d’,‘!’,0
Варианты определения слова со значением 256 (100h) и двойного слова со значением 65539 (10003h).

dw 256	dd 65539
dw 100h	dd 10003h
db 0,1	dw 3, 1
	db 3, 0, 1, 0

Адресное выражение
В качестве операнда директивы DW может быть указано адресное выражение, т.е. выражение, значением которого является адрес. Основной случай адресного выражения — это имя переменной или метки,
например:
В этом случае ассемблер запишет в слово, выделенное под переменную D, адрес переменной C, который становится начальным значением переменной D.
Несколько операндов, конструкция повторения
В правой части директивы DW можно указать любое число операндов, а также конструкцию повторения, например:
E DW 40000, 3 DUP (?).

3.3.Команды передачи данных

3.3.1. Команда MOV

Команда MOV(пересылка = «MOVE operand»). Команда пересылки байта, слова или двойного слова. Пересылаемая величина берется из команды, регистра или ячейки памяти, а записывается в регистр или ячейку памяти. Таких команд много, но в языке ассемблера все они записываются одинаково:
MOV <DEST>,< SRC>

На приведенном рисунке 1 представлены различные способы, которыми в микропроцессоре можно пересылать данные из одного места в другое. Каждый прямоугольник означает регистр или ячейку памяти, которые связаны путями пересылки данных, допускаемыми микропроцессором. Отметим, что все команды микропроцессора могут указывать только один операнд памяти.

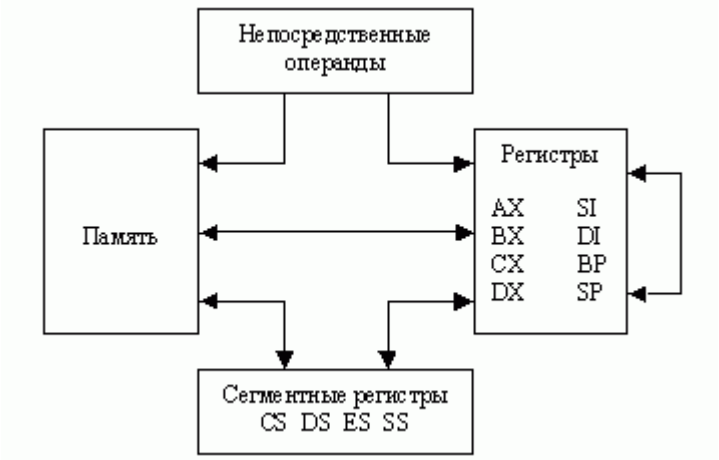

Рис 1 Пересылка данных на микропроцессоре.

Из рисунка видно, что запрещены пересылки из одной ячейки памяти в другую, из одного сегментного регистра в другой, запись непосредственного операнда в память. Это обусловлено тем, что в персональном компьютере отсутствуют соответствующие машинные команды. Если по алгоритму необходимо произвести одно из таких действий, то оно обычно реализуется в две команды, пересылкой через какой-нибудь несегментный регистр. Кроме того, командой MOV нельзя менять содержимое сегментного регистра CS. Это связано с тем, что регистровая пара CS:IP определяет адрес следующей выполняемой команды, поэтому изменение любого из этих регистров есть ничто иное, как операция перехода. Команда же MOV не реализует переход.
Применение:
команда MOV применяется для различного рода пересылок данных, при этом, существуют ограничения и особенности выполнения данной операции:

1. направление пересылки в команде MOV всегда справа налево, то есть из операнда SRC в операнд DEST;
2. значение операнда SRC не изменяется;
3. оба операнда не могут быть из памяти (при необходимости можно использовать цепочечную команду MOVS или сочетание PUSH/POP);
4. лишь один из операндов может быть сегментным регистром;
5. лишь один из операндов может быть управляющим регистром;
6. лишь один из операндов может быть тестовым регистром;
7. лишь один из операндов может быть регистром отладки;
8. лишь один из операндов может быть непосредственным значением;
9. желательно использовать в качестве одного из операндов регистр AL/AX/EAX/RAX, так как в этом случае транслятор генерирует более короткую форму команды MOV.

Команда MOV применяется для обмена данными между системными регистрами. Это одна из немногих возможностей доступа к содержимому этих регистров. Данную команду можно использовать только на нулевом уровне привилегий либо в реальном режиме работы микропроцессора.

Если необходимо переслать в регистр адрес какой-то переменной, то необходимо использовать в команде MOV оператор OFFSET (оператор получения смещения выражения). OFFSET позволяет получить значение смещения выражения в байтах относительно начала того сегмента, в котором выражение определено. А если эта переменная еще и находится в другом сегменте, то в паре с оператором OFFSET требуется использовать оператор SEG (оператор получения сегментной составляющей адреса выражения). Оператор SEG возвращает физический адрес сегмента для выражения, в качестве которого могут выступать метка, переменная, имя сегмента, имя группы или некоторое символическое имя.

3.3.2. Команда LEA

Команда LEA (Загрузка эффективного адреса = "Load Effective Address").
Синтаксис команды:

LEA <DEST>,<SRC>

LEA <DEST>,[<Base>+<Index>*<Scale>+<смещение>],

где <Scale>1,2,4,8

Представим возможные варианты команды:

lea r(16/32/64), mem

Семантика команды представляет следующее - получить в регистр DEST эффективный адрес (смещение) операнда SRC.

Алгоритм работы команды зависит от действующего режима адресации (use 16/use 32 или use 64):

- use 16 - в регистр DEST загружается 16-битное значение смещения операнда SRC ;
- use 32 - в регистр DEST загружается 32-битное значение смещения операнда SRC;
- use 64 - в регистр DEST загружается 64-битное значение смещения операнда SRC.

Псевдокод команды LEA:

```
IF OperandSize = 16 and AddressSize = 16
        THEN DEST <-- EffectiveAddress(SRC); 16-битная адресация
ELSE IF OperandSize = 16 и AddressSize = 32
THEN temp<-- EffectiveAddress(SRC); 32-битная адресация
DEST <-- temp[0:15]; 16-битная адресация
ENDIF;
ELSE IF OperandSize = 32 и AddressSize = 16
THEN temp<-- EffectiveAddress(SRC); 16-битная адресация
DEST <-- ZeroExtend(temp); 32-битная адресация
ENDIF;
ELSE IF OperandSize = 32 и AddressSize = 32
        THEN DEST <-- EffectiveAddress(SRC); 32-битная адресация
ENDIF;
```

ELSE IF OperandSize = 16 и AddressSize = 64
THEN temp<-- EffectiveAddress(SRC); 64-битная адресация
DEST <-- temp[0:15]; 16-битная адресация ENDIF;
ELSE IF OperandSize = 32 и AddressSize = 64
THEN temp<-- EffectiveAddress(SRC); 64-битная адресация DEST <-- temp[0:31]; 16-битная адресация ENDIF;
ELSE IF OperandSize = 64 и AddressSize = 64
THEN DEST<-- EffectiveAddress(SRC); 64- битная адресация ENDIF;
ENDIF;

Отметим, что код эквивалентной команды - mov ebx, offset adr короче, чем код - lea ebx, adr

Таблица 7. Использование команды lea

Для сложения	lea eax,[eax+ebx]=add eax,ebx
Для инкремента	lea eax,[eax+1]
Для декремента	lea eax,[eax-1]
Для умножения на 2	lea eax,[eax+eax] или lea eax,[eax*2]
Для умножения на 3	lea eax,[eax+eax*2]
Для умножения на 4	lea eax,[eax*4]
Для умножения на 5	lea eax,[eax+eax*4]
Для умножения на 6	lea ecx,[eax*2] / lea eax,[ecx+ecx*2]]
Для умножения на 7	lea ecx,[eax+eax*2] / lea eax,[ecx+eax*4]
Для умножения на 8	lea eax,[eax*8]
Для умножения на 9	lea eax,[eax+eax*8]

Команда lea eax,[ebx] эквивалент mov eax,ebx

3.3.3. Команда XCHG

Команда XCHG (Обмен ="EXCHANGE operands")

В программах на языке ассемблера иногда возникает необходимость переставлять местами какие-либо величины, и, хотя такую перестановку можно организовать с помощью двух команд MOV с сохранением значения в промежуточной ячейке памяти, в процессор введена специальная команда для этого. Синтаксис команды: XCHG <SRC>,<DEST>

Семантика команды: обмен значений между двумя регистрами или между регистром и ячейкой памяти.

Алгоритм работы: обмен содержимым операнда SRC и операнда DEST. Псевдокод команды:

TEMP<--DEST
DEST<-- SRC
SRC <--TEMP

Возможные варианты команды:

xchg reg, mem/reg

Применение: команду XCHG можно использовать для выполнения операции обмена двух операндов с целью изменения порядка следования байт, слов, двойных слов или их временного сохранения в регистре или памяти. Альтернативой является использование для этой цели стека или промежуточной ячейки памяти.

Пример:

```
;поменять порядок следования байт в ячейке памяти
H1 DW 0F85Ch ;напрямую командой XCHG нельзя, но
MOV AX,H1 ;можно для этой цели использовать
XCHG AH,AL ;промежуточный регистр AX
MOV H1,AX ;[H1]=5CF8h
```

Команду XCHG можно заменить на XOR или на PUSH/POP, или на MOV с использованием промежуточного регистра или промежуточной ячейки памяти (табл.8).

Таблица 8. Эквиваленты команды XCHG EAX,EBX

MOV [TEMP],EAX	XOR EAX,EBX	PUSH EAX
MOV EAX,EBX	XOR EBX,EAX	PUSH EBX
MOV EBX,[TEMP]	XOR EAX,EBX	POP EAX POP EBX

3.3.4. Команда обмена байтов BSWAP

Команда обмена байтов BSWAP (обмен байтами «BYTE SWAPING») Синтаксис команды: BSWAP <DEST> Возможные варианты команды:

bswap reg(16/32/64)

Семантика команды: команда BSWAP изменяет порядок следования байтов в 32-разрядных регистрах, конвертируя значения из вида машинного представления (младшая часть, старшая часть) в обычное представление (старшая часть, младшая часть) и наоборот. Можно использовать для примитивного шифрования. Псевдокод команды BSWAP:

```
TEMP<--DEST
IF 64-bit mode AND OperandSize = 64
THEN
DEST[7:0]<--TEMP[63:56]
DEST[15:x]<--TEMP[55:4x]
DEST[23:16]<--TEMP[47:40] DEST[31:24]<--TEMP[39:32]
DEST[39:32]<--TEMP[31:24] DEST[47:40]<--TEMP[23:16]
DEST[55:4x]<--TEMP[15:x]
DEST[63:56]<--TEMP[7:0]
ELSE
```

DEST[7:0]<--TEMP[31:24]
DEST[15:x]<--TEMP[23:16]
DEST[23:16]<--TEMP[15:x]
DEST[31:24]<--TEMP[7:0] ENDIF;

Пример1:

mov eax,12345678 h
bswap eax *; eax=78563412h*

Пример2:

Увеличим третий байт регистра EAX на единицу
mov eax,87654321h
bswap eax*; eax=21436587h*
inc al
bswap eax; *eax=88654321h*

Пример3:

Использование BSWAP с 16-разрядными регистрами
mov eax, 12345678h
bswap ax; *eax = 12340000h*

при этой команде содержимое старшей части регистра EAX недоступно (хотя процессор старшую часть регистра всё равно должен видеть) и поэтому происходит обнуление регистра AX.

Оператор указания типа (PTR)

По команде MOV можно переслать как байт и слово, так и двойное слово. Как определить что именно пересылает команда?? Размер пересылаемой величины обычно определяется по типу операндов, указанных в команде MOV.

Пример:

X DB? ; *TYPE X = BYTE*
Y DW? ;TYPE Y = WORD
MOV BH,0; *пересылка байта*
MOV X,0*; пересылка байта (X описан как имя байтовой переменной)*
MOV ESI,0*; пересылка двойного слова*
MOV Y,0 ;*пересылка слова (Y описан как имя переменной-слова)*

Обратите внимание, что по второму операнду (0) нельзя определить, какого он размера: ноль может быть и байтом (00h), и словом (0000h), и двойным словом (00000000h).

Если можно определить размеры обоих операторов, тогда эти размеры должны совпадать (либо байты, либо слова, либо двойные слова), иначе ассемблер зафиксирует ошибку. Пример:

MOV DI,ES ; *пересылка слова*
MOV CH,X ; *пересылка байта*
MOV ESI,AX ;*ошибка (ESI регистр размером в двойное* слово,

;*AX регистр размером в слово)*

MOV BH,300; *ошибка – BH байтовый регистр, а 300 не может быть байтом*

Оператор переопределения типа PTR (от POINTER, указатель) применяется для переопределения или уточнения типа метки или переменной, определяемой выражением. Оператор переопределения типа PTR записывается следующим образом:

<тип> PTR <выражение>

Тип может принимать одно из следующих значений: BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR, а выражение может быть константным или адресным. Оператор PTR используется с атрибутами BYTE, WORD, DWORD и т.д. для локальной отмены определенных типов (DB, DW и т.д.) или с атрибутами NEAR, FAR для отмены значения дистанции по умолчанию. Если указано константное выражение, то использование данного оператора говорит о том, что значение этого выражения (число) должно рассматриваться языком ассемблера, как величина указанного типа (размера); например, BYTE PTR 0 - это ноль как байт, а WORD PTR 0 - это ноль как слово.

Если же указано адресное выражение, то оператор показывает, что адрес, являющийся значением выражения, должен восприниматься языком ассемблера, как адрес ячейки указанного типа (размера); например, WORD PTR A - адрес A обозначает слово (байты с адресами A и A+1). В данном случае оператор PTR относится к адресным выражениям.

3.4. Способы адресации операндов

В программах на Assembler применяются следующие типы адресации операндов: регистровая, прямая, непосредственная, косвенная, базовая, индексная, базово-индексная.

Регистровая адресация подразумевает использование в качестве операнда регистра,

например:

```
push DS
mov BP,SP
```

При прямой адресации один операнд представляет собой адрес памяти, второй регистр, например:

```
move DATA, AX
```

Непосредственная адресация применяется, когда операнд длиной в байт или слово находится в ассемблерной команде, например:

```
mov AX, 4Ch
```

При использовании косвенной адресации исполнительный адрес формируется исходя из сегментного адреса в одном из сегментных регистров и смещения в регистрах BX, BP, SI или DI, например:

mov AL, [BX*]; база находится в регистре DS, смещение в регистре BX*

mov AH, [SI]; *база -в DS, смещение -в SI*

mov AX,[DI*]; база в DS, смещение -в DI*

mov AX, ES: [DI]; *база -в ES, смещение -в DI*

mov DX, [BP]; *база -в SS, смещение -в BP*

В случае применения базовой адресации исполнительный адрес является суммой значения смещения и содержимого регистра BP или BX, например:

mov AX, [BP+6]; *база -SS, смещение -содержимое BP, которое складывается с 6*

mov [BX+Delta], *AX; база -DS, смещение -содержимое BX+смещение Delta*

mov AX, [BP]+4; *база -SS, смещение -содержимое BP+4*

mov DX, x[BX]; *база -DS, смещение -содержимое BX+x*

При индексной адресации исполнительный адрес определяется как сумма значений указанного смещения и содержимого регистра SI или DI так же, как и при базовой адресации, например:

mov DX, [SI+5]; *база -DS, смещение -SI+5*

mov ES:[DI]+6,AL ;*база -ES, смещение -DI+6*

Базово-индексная адресация подразумевает использование для вычисления исполнительного адреса суммы содержимого базового и индексного регистров, а также смещения, находящегося в операторе, например:

mov BX, [BP][SI]

mov ES:[BX+DI],AX

mov Array[BX][SI],12h

mov AX,[BP+6+DI]

mov Array [BP+BX]; *ошибка -два базовых регистра*

mov Array [DI+SI]; *ошибка -два индексных регистра*

Относительная базово-индексная адресация:

mov eax,[edx+ecx+1000h]

В регистр EAX копируется содержимое ячейки памяти, адрес которой вычисляется следующим образом: содержимое регистра Index (=ECX) прибавляется к содержимому регистра Base (=EDX), а к полученному результату добавляется смещение (=1000h). Относительная базово-индексная адресация с масштабированием:mov eax,[edx+ecx*4+1000h]

В регистр EAX копируется содержимое ячейки памяти, адрес которой вычисляется следующим образом: содержимое регистра Index (=ECX) умножается на Scale (может быть 1, 2, 4 и x, в данном случае Scale=4), прибавляется к содержимому регистра Base (=EDX), а к полученному результату добавляется смещение (=1000h).

Косвенная адресация может быть только в одном операнде, но не в двух. Поэтому инструкции, подобные следующей, недопустимы, а при попытке их трансляции будет выдана ошибка: mov [eax],[edx]

3.5. Ввод-вывод

Итак, выводить строки на экран можно двумя путями:

1. Посимвольно (то есть в цикле выводить каждый символ строки).
2. Строку целиком.

В текстовом режиме вывод на экран можно выполнить одним из трёх способов:

1. С помощью функций DOS.
2. С помощью функций BIOS.
3. Путём прямой записи в видеопамять.

Третий способ отличается тем, что он сразу записывает данные в видеопамять, что позволяет выполнять вывод более быстро. Однако в наше время он практически применим, разве что, в учебных целях., так как современные операционные системы не позволяют напрямую обращаться к аппаратуре (реальный режим).

Функции вывода DOS

Прерывание int 21h – команда (диспетчер функций DOS), с помощью которой программа, написанная на языка Ассемблера. запрашивает сервис DOS для выполнения определённых действий типа ввода данных с клавиатуры или их вывод на экран. Программа, запрашивающая сервис DOS, должна подготовить всю необходимую информацию в регистрах и управляющих блоках, указать в регистре AH номер желаемой функции DOS и затем вызвать прерывание INT 21H.

Номера некоторых функций прерывания INT 21H:

• AH = 01: Ввод с клавиатуры с эхо отображением. Данная функция возвращает значение в регистре AL, Если содержимое AL не равно нулю, то оно представляет собой стандартный ASCII-символ, например букву или цифру.

• AH = 02: Вывод символа. Для ввода символа на экран в текущую позицию курсора необходимо поместить код данного символа в регистр DL.

• AH = 06: Ввод/вывод данных. Может использоваться как для ввода, так и для вывода. Для вывода занесите в DL выводимый символ (но не FFH!) и вызовите прерывание 21H. Для ввода в DL занесите FFH, выполните прерывание 21H.

• AH = 09: Вывод строки символов Выводимая строка должна заканчиваться знаком доллара $. Адрес начала строки должен быть помещен в DX. Знак доллара не выводится.

• AH = 0A: Ввод данных в буфер: Определяется максимальная длина вводимого текста. Это необходимо для предупреждения пользователя звуковым сигналом, если набран слишком длинный текст; символы считываются со стандартного ввода вплоть до CR (ASCII 0dH) или до достижения длины MAX-1. если достигнут MAX-1, включается консольный звонок для каждого очередного символа, пока не будет введен возврат каретки CR (нажатие Enter).. Во второй байт буфера команда возвращает действительную длину введенного текста в байтах. Адрес буфе

Листинг

```
.data
        msg DB ‘Hello world$’, 10, 13 ;объявляем строку, которую будем выводить (она обязательно заканчивается знаком "$",
.code
        …
        lea dx, msg;загружаем в dx адрес строки
        mov ah, 09h;загружаем в ah, номер нужной функции
        int 21h;вызываем прерывание
        …
code ends
```

Пример:

```
;=====================================================
; Эта программа выводит на экран строку "Hello, World!!!"
; с помощью функции DOS 09h
;-----------------------------------------------------
        .model      tiny
        .code
        ORG  100h           ;начало

start:
  MOV       AH,   09h          ;Номер функции 09h
  MOV       DX,   offset stroka   ;Адрес строки записываем в DX
  INT 21h
  RET                 ;завершение

strokaDB    'Hello, World!!!$'  ;Строка для вывода
  END       start
;=====================================================
```

При правильном использовании на экран будет выведено примерно следующее:

Рис.2. Вывод на экран с помощью функций DOS.

Функции вывода BIOS

Функции BIOS также могут выводить как отдельные символы (функции 09h, 0Ah, 0Eh), так и строки целиком (функция 13h).

Кроме того, с помощью функций BIOS можно установить видеорежим, установить или считать положение курсора, а также считать символ и его атрибуты.

Хотя функции DOS тоже могут считывать символы, но всё-таки возможности BIOS более широки.

Для работы с функциями BIOS также сначала надо подготовить данные, записать номер функции в регистр AH, а затем вызвать прерывание 10h.

Для примера рассмотрим функцию 13h. Перед вызовом функции надо:

- Записать номер функции в регистр AH.
- Записать режим вывода в регистр AL:
  - бит 0: переместить курсор в конец строки после вывода.
  - бит 1: строка содержит не только символы, но и атрибуты. Так что каждый символ описывается двумя байтами - ASCII-код и атрибут. Это можно использовать, если в строке символы должны иметь, например, разный цвет. Если атрибуты одинаковы для всей строки, то этот бит лучше сбросить (обнулить).
  - биты 2-7: не используются.
- Записать длину строки в регистр CX (только число символов, байты атрибутов не учитываются).
- Если строка содержит только символы, то записать в регистр BL атрибут. Этот атрибут будет применяться для всей строки.
- Записать координаты экрана, начиная с которых будет выводиться строка, в регистры DL (столбец - координата X) и DH (строка - координата Y).
- Записать адрес начала строки в ES:BP.

Пример:

```
;=========================================================
; Эта программа выводит на экран строку "Hello, World!!!"
; с помощью функции BIOS 13h
;---------------------------------------------------------
        .model      tiny
        .code
        ORG 100h        ;начало

start:
  MOV   AH, 13h         ;Номер функции 13h
  MOV   AL, 1           ;Перевести курсор в конец строки
  MOV   CX, 15          ;Длина строки
  MOV   BL, 00011110b   ;Жёлтый текст на синем фоне
  MOV   DL, 5           ;Координата X
  MOV   DH, 2           ;Координата Y
```

```
 MOV    BP,  offset stroka   ;Адрес строки записываем в DX
 INT    10h
 RET                         ;завершение

stroka   DB  'Hello, World!!!' ;Строка для вывода
 END    start
;=============================================================
```

Рис.3. Вывод на экран с помощью функций BIOS.

4. Задание на выполнение

Варианты заданий

1. Задана последовательность чисел произвольной длины из 100 символов. Числа больше 10 заменить на 9. Вывести на экран количество таких чисел в каждой строке и запомнить их в новую последовательность.

2. Задан текст из 100 символов, содержащий слова произвольной длины. Слова в тексте разделены пробелами. Создать новый текст, в который поместить количество букв в каждом слове. Вывести на экран количество слов в тексте.

3. Задан текст из 100 символов, содержащий слова произвольной длины. Слова в тексте разделены пробелами. Создать новый текст, в который поместить количества букв в каждом слове. Вывести на экран все слова текста, каждое с новой строки.

Выполнение лабораторной работы

1. Напишите программу на Ассемблере в соответствии с заданием.
2. Выполните ассемблирование.
3. Распечатайте листинг программы.
4. Распечатайте результаты работы программы.

5. Подготовьте отчет по лабораторной работе.

Контрольные вопросы

1. Какие методы адресации предоставляет Ассемблер?
2. Особенности прерывания int 21.
3. Особенности прерывания int 10.
4. Особенности прерывания int 16.

Литература

1.В.И.Юров .- Ассемблер, 2 издание- СПб: Питер, 2003.
2.Пирогов В. Ю.- Ассемблер для Windows. Изд. 4-е перераб. и доп. — СПб.: БХВ- Петербург, 2015.
3. И.А. Калашников. Ассемблер – это просто. Программирование на Ассемблере. «Бином», 200х г.

Приложение 1.

Листинг программы, которая осуществляет ввод строки с клавиатуры и заменяет все буквы «а» на заглавные, пробелы удаляются.
Файл inpstr.asm:

```
.model tiny
.code
org 100h
start:
;выводим приглашение
mov dx,offset message
mov ah,09h
int 21h
;вводим строку
mov ah,0Ah
mov dx,offset inpstr
int 21h
;выводим то что ввели
mov dx,offset inpstr+2
mov ah,09h
int 21h
;начинаем цикл обработки
;с индекса 2 в массиве начинается введенная информация
mov si,2
;нулевое смещение для преобразованной строки
```

```
mov di,0
;по индексу 1 в массиве находится длинна
mov cl, byte ptr [inpstr+1]
;теперь в cx длинна - для цикла
mov ch,0
;сам цикл
lop: mov ah,inpstr[si]
;сравниваем элемент строки с буквой "a"
cmp ah,'a'
jne met
;пишем в новую строку заглавную A
mov dl,'A'
mov sn[di],dl
;увеличиваем индекс для новой строки
inc di
jmp met1
;пробелы пропускаем!
met: cmp ah,' '
je met1
;это не «a» и не пробел- поэтому просто переносим букву в новую строку
mov sn[di],ah
inc di
;увеличим индекс для следующего элемента исходной строки
met1: inc si
;цикл пока не переберём все элементы строки
loop lop
;выводим сообщение
mov dx,offset message2
mov ah,09h
int 21h
;выводим исправленную строку
mov dx,offset sn
mov ah,09h
int 21h
ret
message db 'Please, input string:',0Ah,0Dh,'$'
message2 db 'It is new string:',0Ah,0Dh,'$'
;строка, которую мы введём
inpstr db 10,?,10 dup ('$'),0Ah,0Dh,'$'
;новая строка
sn db 10 dup ('$'),0Ah,0Dh,'$'
end start
```

Приложение 2

Листинг программы.

Вывод числа с основанием cc

Число в ax

OutInt proc

```
;; если число знаковое, то необходимо расскоментировать следующие строки
;; Проверяем число на знак.
;   test    ax, ax
;   jns     oi1
;
;; Если оно отрицательное, выведем минус и оставим его модуль.
;   mov  cx, ax
;   mov     ah, 02h
;   mov     dl, '-'
;   int     21h
;   mov  ax, cx
;   neg     ax
;; Количество цифр будем держать в CX.
;oi1:
    xor     cx, cx
    mov     bx, 10 ; основание cc. 10 для десятеричной и т.п.
oi2:
    xor     dx,dx
    div     bx
; Делим число на основание cc. В остатке получается последняя цифра.
; Сразу выводить её нельзя, поэтому сохраним её в стэке.
    push    dx
    inc     cx
; А с частным повторяем то же самое, отделяя от него очередную
; цифру справа, пока не останется ноль, что значит, что дальше
; слева только нули.
    test    ax, ax
    jnz     oi2
; Теперь приступим к выводу.
    mov     ah, 02h
oi3:
    pop     dx
; Извлекаем очередную цифру, переводим её в символ и выводим.
;; раскоментировать если основание cc > 10, т.е. для вывода требуются буквы
;   cmp     dl,9
;   jbe     oi4
;   add     dl,7
;oi4:
```

```
    add     dl, '0'
    int     21h
; Повторим ровно столько раз, сколько цифр насчитали.
    loop    oi3

    ret
```

беззнаковое 0-99

```
OutInt proc
    aam
    add ax,3030h
    mov dl,ah
    mov dh,al
    mov ah,02
    int 21h
    mov dl,dh
    int 21h
OutInt endp

16 система счисления
byte2hex        proc    near
                push    cx
                mov     cx,2
@@L1:           rol     dl,4
                mov     ax,300fh
                and     al,dl
                aaa
                aad     11h
                stosb
                loop    @@L1
                pop     cx
                ret
byte2hex        endp
 word2hex       proc    near
                push    cx
                mov     cx,2
@@L1:           rol     dx,x
                call    byte2hex
                loop    @@L1
                pop     cx
                ret
word2hex        endp

dword2hex       proc    near
                mov     cx,2
```

```
@@L1:          rol     edx,16
               call    word2hex
               loop    @@L1
               ret
dword2hex      endp
```

Двоичная система счисления

OutBin proc

```
 mov bx,ax
   mov cx,16
ob1:
   shl bx,1
   jc ob2

   mov dl,'0'
   jmp ob3

ob2:
   mov dl,'1'
ob3:
   mov ah,2
   int 21h
   loop ob1

OutBin endp
```

Ввод целого числа

InputInt proc

```
    mov ah,0ah
   xor di,di
   mov dx,offset buff ; аддрес буфера
   int 21h ; принимаем строку
   mov dl,0ah
   mov ah,02
   int 21h ; выводим перевода строки

; обрабатываем содержимое буфера
   mov si,offset buff+2 ; берем аддрес начала строки
   cmp byte ptr [si],"-" ; если первый символ минус
   jnz ii1
   mov di,1  ; устанавливаем флаг
   inc si    ; и пропускаем его
ii1:
   xor ax,ax
   mov bx,10  ; основание сс
ii2:
```

```
    mov cl,[si] ; берем символ из буфера
    cmp cl,0dh  ; проверяем не последний ли он
    jz endin

; если символ не последний, то проверяем его на правильность
    cmp cl,'0'  ; если введен неверный символ <0
    jb er
    cmp cl,'9'  ; если введен неверный символ >9
    ja er

    sub cl,'0' ; делаем из символа число
    mul bx     ; умножаем на 10
    add ax,cx  ; прибавляем к остальным
    inc si     ; указатель на следующий символ
    jmp ii2    ; повторяем

er:   ; если была ошибка, то выводим сообщение об этом и выходим
    mov dx, offset error
    mov ah,09
    int 21h
    int 20h

; все символы из буфера обработаны число находится в ax
endin:
    cmp di,1 ; если установлен флаг, то
    jnz ii3
    neg ax   ; делаем число отрицательным
ii3:
    ret

error db "incorrect number$"
buff    db 6,7 Dup(?)
InputInt endp
```

Вывод .2x6C

```
.model small
.data
 flt_num dq      -324.73412347x1
 .code
start:
mov ax, @data
mov ds, ax
 finit
fld flt_num
```

```
push 10
call outfloat
 .exit
 ; Вывод вещественного числа
; аргумент - количество цифр дробной части
length_frac     equ     [bp+4]
; локальные переменные
ten     equ word ptr [bp-2]
temp    equ word ptr [bp-4]

OutFloat proc   near
        enter   4, 0            ; пролог - выделим в кадре стека 4 байта под локальные
переменные
        mov     ten, 10
        ftst                    ; определяем знак числа
        fstsw   ax
        sahf
        jnc     @positiv
        mov     al, '-'         ; если число отрицательное - выводим минус
        int     29h
        fchs                    ; и получаем модуль числа
@positiv:
        fld1                    ; загружаем единицу
        fld     st(1)           ; копируем число на вершину стека
        fprem                   ; выделим дробную часть
        fsub    st(2), st       ; отнимем ее от числа - получим целую часть
        fxch    st(2)           ; меняем местами целую и дробную части
        xor     cx, cx          ; обнуляем счетчик
; далее идет стандартный алгоритм вывода целого числа на экран
@1:
        fidiv   ten             ; делим целую часть на десять
        fxch    st(1)           ; обменяем местами st и st(1) для команды fprem
        fld     st(1)           ; копируем результат на вершину стека
        fprem                   ; выделим дробную часть (цифру справа от целой части)
        fsub    st(2), st       ; получим целую часть
        fimul   ten             ; *10
        fistp   temp            ; получаем очередную цифру
        push    temp            ; заталкиваем ее глубже в стек
        inc     cx              ; и увеличим счетчик
        fxch    st(1)           ; подготовим стек к следующему шагу цикла (полученное
частное на вершину, в st(1) - 1)
        ftst                    ; проверим не получили ли в частном 0?
        fstsw   ax
        sahf
```

```
    jnz    @1            ; нет - продолжим цикл
@2:                       ; извлекаем очередную цифру, переводим её в символ и выводим.
    pop    ax
    add    al, '0'
    int    29h
    loop   @2
; далее то же самое, только для дробной части. Алгоритм похож на вывод целого числа, только вместо деления умножение и проход по числу слева
    fstp   st            ; сначала проверим, есть ли дробная часть
    fxch   st(1)
    ftst
    fstsw  ax
    sahf
    jz     @quit         ; дробная часть отсутствует
    mov    al, '.'
    int    29h           ; если присутствует - выведем точку
    mov    cx, length_frac ; помещаем в счетчик длину дробной части
@3:
    fimul  ten           ; умножим на 10
    fxch   st(1)         ; подготовка для fprem - меняем st и st(1) местами и
    fld    st(1)         ; копируем число на вершину
    fprem                ; отделим дробную часть от целой
    fsub   st(2), st     ; и оставляем дробную
    fxch   st(2)
    fistp  temp          ; выталкиваем полученное число из стека в temp
    mov    ax, temp      ; по дробной части идем слева, значит число выводим сразу, без предварительного сохранения в стек
    or     al, 30h       ; перевод в ascii
    int    29h           ; на экран
    fxch   st(1)         ; подготовим стек к следующему шагу цикла (полученное частное на вершину, в st(1) - 1)
    ftst
    fstsw  ax
    sahf                 ; проверим на 0 остаток дробной части
    loopne @3
@quit:
    fstp                 ; готово. Чистим стек сопроцессора
    fstp   st
    leave                ; эпилог
    ret    2
OutFloat endp
 end start
```