ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

ОРГАНИЗАЦИЯ ЭВМ И ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

Методические указания к лабораторному практикуму

«Основы языка Assembler»

Самара, 2013

СОДЕРЖАНИЕ

1 Арифметические и логические команды в ассемблере 6
1.1 Арифметические команды 6
1.1.1 Сложение и вычитание 6
1.1.2 Переполнения 6
1.1.3 Беззнаковые и знаковые данные 7
1.1.4 Умножение 8
1.1.5 Беззнаковое умножение: Команда MUL 9
1.1.6 Знаковое умножение: Команда IMUL 9
1.1.7 Многословное умножение 10
1.1.8 Деление 11
1.1.9 Беззнаковое деление: Команда DIV 12
1.1.10 Переполнения и прерывания 13
1.1.11 Преобразование знака 13
1.2 Логические команды 14
1.2.1 Команды логических операций : and, not,or,xor,test 14
1.2.2 Команды сдвига и циклического сдвига 15
1.3 Примеры 18
1.4 Варианты заданий 3
2.1 Получение символов с клавиатуры 3
2.2 Вывод символов на экран 3
2.3 Безусловные переходы 4
2.4 Условные переходы 6
2.5 Пример 9
2.6 Задания 11
3 Программирование на языке ассемблер задач с использованием массивов строковых данных 14
3.1 Предопределенный идентификатор $ 14
3.2 Цепочные команды 15
3.2.1 Инструкция LODS 15

3.2.2 Инструкция STOS 16
3.2.3 Инструкция MOVS 18
3.2.4 Повторение строковой инструкции 19
3.2.4 Сравнение строк 20
3.3 Режимы адресации к памяти 21
3.4 Ввод-вывод 31
3.6 Задания 34
4 Работа с массивами и стеком на языке ассемблера 36
4.1 Общие сведения о массивах 36
4.2 Ввод – вывод массива 36
4.3 Способы сортировки массивов. 38
4.4 Работа со стеком в ассемблере 42
4.4.1 Команды работы со стеком 42
4.4.2 Передача параметров в стеке 44
4.4.3 Передача параметров в потоке кода 46
4.5 Задания 47
5 Работа с математическим сопроцессором в среде Assembler 50
5.1 Основные сведения 50
5.2 Команды сопроцессора 52
5.2.1 Команды пересылки данных 52
5.2.2 Арифметические команды 52
5.3 Пример 57
5.4 Задания 58
6 Программирование на языке ассемблера задач с использованием системных ресурсов BIOS. Работа в графическом режиме 61
6.1 Графический режим 61
6.2 Прерывание BIOS INT 10H для графики 62
6.3 Задания 63
7 Работа с файлами в языке Assembler 65
7.1 Создание файла 65

7.2 Открытие существующего файла 66
7.3 Создание и открытие файла. 67
7.4 Чтение, запись и переименование файла 69
7.5 Перемещение указателя чтения/записи 69
7.6 Запись в файл или устройство 70
7.7 Переименование файла 71
7.8 Закрытие и удаление файла 72
7.8.1 Закрыть файл 72
7.8.2 Удаление 72
7.9 Удаление файлов с длинным именем 73
7.10 Поиск файлов 73
7.10.1 Найти первый файл 73
7.10.2 Найти следующий файл 75
7.11 Задания 76
Список литературы 78

1 АРИФМЕТИЧЕСКИЕ И ЛОГИЧЕСКИЕ КОМАНДЫ В АССЕМБЛЕРЕ

1.1 Арифметические команды

1.1.1 Сложение и вычитание

Команды ADD и SUB выполняют сложение и вычитание байтов или слов, содержащих двоичные данные. Вычитание выполняется в компьютере по методу сложения с двоичным дополнением: для второго операнда устанавливаются обратные значения бит и прибавляется 1, а затем происходит сложение с первым операндом. Во всем, кроме первого шага, операции сложения и вычитания идентичны.

Примеры показывают все пять возможных ситуаций:

- сложение/вычитание регистр-регистр;

add ax,bx

add ah,al

- сложение/вычитание память-регистр;

sub [000ah],ax

- сложение/вычитание регистр-память;

add ax, [000ah]

- сложение/вычитание регистр- непосредственное значение;

add ax,279

sub ch,3

- сложение/вычитание память - непосредственное значение.

sub cs:[000ah],3

Поскольку прямой операции память-память не существует, данная операция выполняется через регистр.

1.1.2 Переполнения

Опасайтесь переполнений в арифметических операциях. Один байт содержит знаковый бит и семь бит данных, т. е. значения от -128 до +127. Результат

арифметической операции может легко превзойти емкость однобайтового регистра. Например, результат сложения в регистре AL, превышающий его емкость, автоматически не переходит в регистр AH. Предположим, что регистр AL содержит 60_{16}, тогда результат команды **Add AL, 20H** генерирует AL сумму - 80_{16}. Но операция также устанавливает флаг переполнения и знаковый флаг в состояние "отрицательно". Причина заключается в том, что 80_{16} или двоичное 1000 0000 является отрицательным числом. Т.е. в результате, вместо +128, мы получим -128. Так как регистр AL слишком мал для такой операции и следует воспользоваться регистром AX. Но полное слово имеет также ограничение: один знаковый бит и 15 бит данных, что соответствует значениям от -32768 до +32767.

1.1.3 Беззнаковые и знаковые данные

Для беззнаковых величин все биты являются битами данных, и вместо ограничения +32767 регистр может содержать числа до +65535. Для знаковых величин левый байт является знаковым битом. *Команды **ADD** и **SUB** не делают разницы между знаковыми и беззнаковыми величинами, они просто складывают и вычитают биты*. В следующем примере сложения двух двоичных чисел, первое число содержит единичный левый бит. Для беззнакового числа биты представляют положительное число 249, для знакового - отрицательное число -7:

Двоичное представление	Беззнаковое	Знаковое
11111001	249	-7
00000010	2	+2
11111011	251	-5

Двоичное представление результата сложения одинаково для беззнакового и знакового числа. Однако, биты представляют +251 для беззнакового числа и -5 для знакового. Таким образом, числовое содержимое поля может интерпретироваться по-разному.

Состояние "перенос" возникает в том случае, когда имеется перенос в знаковый разряд. Состояние "переполнение" возникает в том случае, когда перенос в знаковый разряд не создает переноса из разрядной сетки или перенос из разрядной сетки

происходит без переноса в знаковый разряд. При возникновении переноса при сложении беззнаковых чисел, результат получается неправильный:

Двоичное представление	Беззнаковое	Знаковое	CF (перенос)	OF (переполнение)
A=11111100	252	-4		
B=00000101	5	+5		
A+B=00000001 неверно	1	1	1	0

При возникновении переполнения при сложении знаковых чисел, результат получается неправильный:

Двоичное представление	Беззнаковое	Знаковое	CF (перенос)	OF (переполнение)
A=01111001	121	+121		
B=00001011	11	+11		
A+B=10000100 неверно	132	-124	0	1

При операциях сложения и вычитания может одновременно возникнуть и переполнение, и перенос:

Двоичное представление	Беззнаковое	Знаковое	CF (перенос)	OF (переполнение)
A=11110110	246	-10		
B=10001001	137	-119		
A+B=01111111 неверно	127	+127	1	1

1.1.4 Умножение

Операция умножения для беззнаковых данных выполняется командой **MUL**, а для знаковых - **IMUL**. Ответственность за контроль над форматом обрабатываемых чисел и за выбор подходящей команды умножения лежит на самом программисте. Существуют две основные операции умножения:

«Байт на байт». Один из множителей находится в регистре AL, а другой в байте памяти или в однобайтовом регистре. После умножения произведение находится в регистре AX. Операция игнорирует и стирает любые данные, которые находились в регистре AH.

«Слово на слово». Один из множителей находится в регистре AX, а другой - в слове памяти или в регистре. После умножения произведение находится в двойном слове, для которого требуется два регистра: старшая (левая) часть произведения находится в регистре DX, а младшая (правая) часть в регистре AX. Операция игнорирует и стирает любые данные, которые находились в регистре DX.

В единственном операнде команд MUL и IMUL указывается множитель. Рассмотрим следующую команду:

Mul multr

Если поле MULTR определено как байт (DB), то операция предполагает умножение содержимого AL на значение байта из поля MULTR. Если поле MULTR определено как слово (DW), то операция предполагает умножение содержимого AX на значение слова из поля MULTR. Если множитель находится в регистре, то длина регистра определяет тип операции, как это показано ниже:

Mul cl ;Байт-множитель: множимое в AL, произведение в AX

Mul bx ;Слово- множитель: множимое в AX, произведение в DX:AX

1.1.5 Беззнаковое умножение: Команда MUL

Команда MUL умножает беззнаковые числа.

Пример:

```
n db 10
. . .
mov al,2
mul n ;ax=2*10=20=0014h: ah=00h al=14h
mov al,26
mul n ;ax=26*10=260=0104h: ah=01h al=04h
```

1.1.6 Знаковое умножение: Команда IMUL

Команда IMUL умножает знаковые числа.

Пример:

```
mov ax,8
mov bx,-1
imul bx ; dx:ax=-8=0fffffff8h=0014h: dx=0ffffh ax=0fff8h
```

Таким образом, если множимое и множитель имеет одинаковый знаковый бит, то команды MUL и IMUL генерируют одинаковый результат. Но, если сомножители имеют разные знаковые биты, то команда MUL вырабатывает положительный результат умножения, а команда IMUL - отрицательный.

Повышение эффективности умножения: При умножении на степень числа 2 (2,4,8 и т.д.) более эффективным является сдвиг влево на требуемое число битов. Сдвиг более чем на 1 требует загрузки величины сдвига в регистр CL. В следующих примерах предположим, что множимое находится в регистре AL или AX:

Умножение на 2: shl ax,1

Умножение на 8: mov ax,3

Shl ax,cl

1.1.7 Многословное умножение

Обычно умножение имеет два типа: "байт на байт" и "слово на слово". Как уже было показано, максимальное знаковое значение в слове ограничено величиной +32767. Умножение больших чисел требует выполнения некоторых дополнительных действий. Рассматриваемый подход предполагает умножение каждого слова отдельно и сложение полученных результатов.

Рассмотрим следующее умножение в десятичном формате:

```
      1  3  6  5
x           1  2
-----------------
+     2  7  3  0
      1  3  6  5
-----------------
      1  8  3  8  0
```

Представим, что десятичная арифметика может умножать только двухзначные числа. Тогда можно умножить 13 и 65 на 12 раздельно, следующим образом:

		1	3				6	5
	x	1	2			x	1	2
	+	2	6			+1	3	0
	1	3				6	5	
	1	5	6			7	8	0

Следующим шагом сложим полученные произведения, но поскольку число 13 представляло сотни, то первое произведение в действительности будет 15600:

	1	5	6	0	0
+			7	8	0
1	6	3	8	0	

Ассемблерная программа использует аналогичную технику, за исключением того, что данные имеют размерность слов (четыре цифры) в шестнадцатеричном формате.

Умножение двойного слова на слово (z=x*y).

Пример:

```
X dd ?
Y dw ?
Z dw ? , ? , ?
. . .
wp equ word ptr
mov ax,wp x ;
mul y
mov z,ax
mov bx,dx
mov ax,wp x+2
mul y
add ax,bx
mov z+2,ax
adc dx,0
mov z+4,dx
```

1.1.8 Деление

Операция деления для беззнаковых данных выполняется командой **DIV**, а для знаковых - **IDIV**. Ответственность за подбор подходящей команды лежит на программисте. Существуют две основные операции деления:

Деление *«Слово на байт»*. Делимое находится в регистре AX, а делитель - в байте памяти или в однобайтовом регистре. После деления остаток получается в регистре AH, а частное - в AL. Так как однобайтовое частное очень мало (максимально+255 (FF_{16}) для беззнакового деления и +127 ($7F_{16}$) для знакового), то данная операция имеет ограниченное использование.

Деление *«Двойное слово на слово».* Делимое находится в регистровой паре DX:AX, а делитель - в слове памяти или в регистре. После деления остаток получается в регистре DX, а частное в регистре AX. Частное в одном слове допускает максимальное значение +32767 ($FFFF_{16}$) для беззнакового деления и +16383 ($7FFF_{16}$) для знакового.

В единственном операнде команд DIV и IDIV указывается делитель. Рассмотрим следующую команду:

div divisor

Если поле DIVISOR определено как байт (DB), то операция предполагает деление слова на байт. Если поле DIVISOR определено как слово (DW), то операция предполагает деление двойного слова на слово.

При делении, например, 13 на 3, получается результат 4 1/3. Частное есть 4, а остаток - 1. Заметим, что ручной калькулятор (или программа на языке BASIC) выдает в этом случае результат 4,333.... Значение содержит целую часть (4) и дробную часть (,333). Значение 1/3 и 333... есть дробные части, в то время как 1 есть остаток от деления.

1.1.9 Беззнаковое деление: Команда DIV

Команда DIV делит беззнаковые числа.

Пример

```
Mov ax,100
Mov bh,2
Div bh ; 100 div 2=50, ah=0 al=50
```

1.1.10 Переполнения и прерывания

Используя команды **DIV** и особенно **IDIV**, очень просто вызвать переполнение. Прерывания приводят (по крайней маре в системе, используемой при тестировании этих программ) к непредсказуемым результатам. В операциях деления предполагается, что частное значительно меньше, чем делимое. Деление на ноль всегда вызывает прерывание. Но деление на 1 генерирует частное, которое равно делимому, что может также легко вызвать прерывание.

Рекомендуется использовать следующее правило: если делитель - байт, то его значение должно быть меньше, чем левый байт (AH) делителя: если делитель - слово, то его значение должно быть меньше, чем левое слово (DX) делителя.

Проиллюстрируем данное правило для делителя, равного 1:

Операция деления: Делимое Делитель Частное

Слово на байт: 0123 01 (1)23

Двойное слово на слово: 0001 4026 0001 (1)4026

В обоих случаях частное превышает возможный размер. Для того чтобы избежать подобных ситуаций, полезно вставлять перед командами DIV и IDIV соответствующую проверку.

Для команды IDIV данная логика должна учитывать тот факт, что либо делимое, либо делитель могут быть отрицательными, а так как сравниваются абсолютные значения, то необходимо использовать команду NEG для временного перевода отрицательного значения в положительное.

1.1.11 Преобразование знака

Команда **NEG** обеспечивает преобразование знака двоичных чисел из положительного в отрицательное и наоборот.

Практически команда NEG устанавливает противоположные значения битов и прибавляет 1.

Примеры:

```
Neg ax
Neg bl
```

Neg BINAMT (байт или слово в памяти)

Преобразование знака для 35-битового (или большего) числа включает больше шагов. Предположим, что регистровая пара DX:AX содержит 32-битовое двоичное число. Так как команда NEG не может обрабатывать два регистра одновременно, то ее использование приведет к неправильному результату. В следующем примере показано использование команды NOT:

```
Not dx ;Инвертирование битов
Not ax ;Инвертирование битов
Add ax,1 ;Прибавление 1 к AX
Adc dx,0 ;Прибавление переноса к DX
```

1.2 Логические команды

1.2.1 Команды логических операций : and, not,or,xor,test

Логические операции являются важным элементом в проектировании микросхем и имеют много общего в логике программирования. Команды **AND**, **OR**, **XOR** и **TEST** являются командами логических операций. Эти команды используются для сброса и установки бит и для арифметических операций в коде ASCII . Все эти команды обрабатывают один байт или одно слово в регистре или в памяти, и устанавливают флаги ***CF, OF, PF, SF, ZF***.

AND: Если оба из сравниваемых битов равны 1, то результат равен 1; во всех остальных случаях результат - 0.

```
Mov al,00110011b
And al,11101110b ; al=0010010b
```

OR: Если хотя бы один из сравниваемых битов равен 1, то результат равен 1; если сравниваемые биты равны 0, то результат - 0.

```
mov al,00110011b
or al,11101110b ; al=11111111b
```

XOR: Если один из сравниваемых битов равен 0, а другой равен 1, то результат равен 1; если сравниваемые биты одинаковы(оба - 0 или оба - 1) то результат - 0.

```
mov al,00110011b
xor al,11101110b ; al=11011101b
```

TEST: действует как AND-устанавливает флаги, но не изменяет биты.

```
mov al,00110011b
test al,11101110b ; al=0011011b sf=0
```

Первый операнд в логических командах указывает на один байт или слово в регистре или в памяти и является единственным значением, которое может изменятся после выполнения команд.

1.2.2 Команды сдвига и циклического сдвига

Команды сдвига и циклического сдвига, которые представляют собой часть логических возможностей компьютера, имеют следующие свойства:

- обрабатывают байт или слово;
- имеют доступ к регистру или к памяти;
- сдвигают влево или вправо;
- сдвигают на величину до 8 бит (для байта) и 16 бит (для слова);
- сдвигают логически (без знака) или арифметически (со знаком).

Значение сдвига на 1 может быть закодировано как непосредственный операнд, значение больше 1 должно находиться в регистре CL.

Команды сдвига

При выполнении команд сдвига флаг CF всегда содержит значение последнего выдвинутого бита. Существуют следующие команды сдвига:

SHR	Логический (беззнаковый) сдвиг вправо
SHL	Логический (беззнаковый) сдвиг влево
SAR	Арифметический сдвиг вправо
SAL	Арифметический сдвиг влево

Следующий фрагмент иллюстрирует выполнение команды **SHR**:

```
Mov cl,03 ; AX:
Mov ax,10110111B ; 10110111
Shr ax,1 ; 01011011 ;Сдвиг вправо на 1
Shr ax,cl ; 00001011 ;Сдвиг вправо на 3
```

Первая команда SHR сдвигает содержимое регистра AX вправо на 1 бит. Выдвинутый в результате один бит попадает в флаг CF, а самый левый бит регистра AX заполняется нулем. Вторая команда сдвигает содержимое регистра AX еще на три бита. При этом флаг CF последовательно принимает значения 1, 1, 0, а в три левых бита в регистре AX заносятся нули.

Рассмотрим действие команд арифметического вправо **SAR**:

```
Mov cl,03 ; AX:
Mov ax,10110111B ; 10110111
Sar ax,1 ; 11011011 ;Сдвиг вправо на 1
Sar ax,cl ; 11111011 ;Сдвиг вправо на 3
```

Команда SAR имеет важное отличие от команды SHR: для заполнения левого бита используется знаковый бит. Таким образом, положительные и отрицательные величины сохраняют свой знак.

В приведенном примере знаковый бит содержит единицу. При сдвигах влево правые биты заполняются нулями. Таким образом, результат команд сдвига SHL и SAL индентичен. Сдвиг влево часто используется для удваивания чисел, а сдвиг вправо - для деления на 2. Эти операции осуществляются значительно быстрее, чем команды умножения или деления.

Деление пополам нечетных чисел (например, 5 или 7) образует меньшие значения (2 или 3, соответственно) и устанавливают флаг CF в 1. Кроме того, если необходимо выполнить сдвиг на 2 бита, то использование двух команд сдвига более эффективно, чем использование одной команды с загрузкой регистра CL значением 2.

Команды циклического сдвига

Циклический сдвиг представляет собой операцию сдвига, при которой выдвинутый бит занимает освободившийся разряд.

Существуют следующие команды циклического сдвига:

ROR	Циклический (беззнаковый) сдвиг вправо
ROL	Циклический (беззнаковый) сдвиг влево

RCR	Циклический сдвиг вправо с переносом
RCL	Циклический сдвиг влево с переносом

Следующая последовательность команд иллюстрирует операцию циклического сдвига **ROR**:

```
Mov cl,03 ; BX:
Mov bx,10110111B ; 10110111
Ror bx,1 ; 11011011 ;Сдвиг вправо на 1
Rorbx,cl ; 01111011 ;Сдвиг вправо на 3
```

Первая команда ROR при выполнении циклического сдвига переносит правый единичный бит регистра BX в освободившуюся левую позицию. Вторая команда ROR переносит, таким образом, три правых бита.

В командах RCR и RCL в сдвиге участвует флаг CF. Выдвигаемый из регистра бит заносится в флаг CF, а значение CF при этом поступает в освободившуюся позицию.

Рассмотрим пример, в котором используются команды циклического и простого сдвига. Предположим, что 32-битовое значение находится в регистрах DX:AX так, что левые 16 бит лежат в регистре DX, а правые - в AX. Для умножения на 2 этого значения возможны следующие две команды:

```
Shl ax,1 ;Умножение пары регистров
Rcl dx,1 ; DX:AX на 2
```

Здесь команда SHL сдвигает все биты регистра AX влево, причем самый левый бит попадает в флаг CF. Затем команда RCL сдвигает все биты регистра DX влево и в освободившийся правый бит заносит значение из флага CF.

1.3 Примеры

Пример 1

Вычислить значение уравнения $y = \frac{12 + 3}{8 + 6} * 3 + 12$

Решение:

```
data segment
y db 0 ; описание переменной y в сегменте данных
data ends
st segment stack ‘stack’ ; описание сегмента стека
db 128 dup(?)
st ends
assume cs: code, ds: data, ss: st
code segment ;описание кодового сегмента
start:
mov ax,ds ; инициализация сегмента данных
mov ds,ax
mov ax,12 ; реализация сложения 12+3
add ax,3
mov bl,8 ; реализация сложения 8+6
add bl,6
div bl ;делим содержимое ax с содержимым bl
mov ah,0 ; остаток обнуляем и результат умножаем на 3
mov bl,3
mul bl
add ax,12 ; к произведению прибавляем 12 и заносим в y
mov y,al
mov ax,4c00h ;завершаем работу программы
int 21h
code ends
end start
```

Пример 2

Даны два числа в двоичном виде. Первое число проинвертировать и разделить на 4. второе число умножить на 2. Результаты логически сложить и первые четыре разряда заменить на противоположные.

Решение:

```
data segment
a db 10110101b
b db 00110111b
c db 0
data ends
st segment stack ‘stack’ ; описание сегмента стека
db 128 dup(?)
st ends
assume cs: code, ds: data, ss: st
code segment ;описание кодового сегмента
start:
mov ax,ds ; инициализация сегмента данных
mov ds,ax
not a ; инвертируем первое число и делим его на 4
shr a,2
shl b,1 ; второе число умножаем на 2
mov al , a ; полученные результаты складываем
add al , b
xor al, 00001111b ; меняем первые четыре разряда на противоположные
mov c,al
mov ax,4c00h ;завершаем работу программы
int 21h
code ends
end start
```

1.4 Варианты заданий

№ варианта	Задание 1 Вычислить значение уравнения	Задание 2
1.	$y = \frac{12 / 5 + 15 / 9}{8 + 6} + \frac{13 / 3 - 2 / 4}{3 - 2}$	Дано число в двоичном виде. Умножить его на 16. Результат перевернуть следующим образом: нулевой разряд становится седьмым, 1-ый становится 6-ым и т.д.
2.	$y = \frac{25}{13} - \frac{12 / 3 - (12 + 4) / 2}{5 + 2}$	Даны два числа в двоичном виде. В первом числе 3,5,7 разряды обнулить и результат разделить на 4, полученное значение логически умножить на 2-ое число.
3.	$y = \frac{16 / 3 - 8 * 2 + 3 * 5}{15 / 6}$	Дано число в двоичном виде. Поменять местами старшую и младшую части числа. Полученное значение разделить на 32 и проинвертировать.
4.	$y = \frac{5 + 9}{3} - \frac{16 * 3}{2 + 3} + \frac{23 / 3}{3} - 2$	Даны два числа в двоичном виде. В первом числе старшие (4 разряда) разряды обнулить. Во втором числе сделать единицами 2,4,6 разряды. Полученные результаты логически перемножить.
5.	$y = \frac{33 / 13 + 12 * 2}{3 * 4} - \frac{4 * 6}{5}$	Дано число в двоичном виде. Разделить его на 16, занести в 1,3,7 разряды нули. Полученное значение логически сложить с числом 19.
6.	$y = \frac{11 * 2 - 6 * 4}{15 - 7} - \frac{29 / 4 + 7 / 4}{2 + 3}$	Даны два числа в двоичном виде. Первое число умножить на 9, второе разделить на 4. результаты логически перемножить и старшую часть поменять местами с младшей.
7.	$y = \frac{(22 / 3 - 13 / 5 + 3) / 2}{5 + 6} - 12$	Дано число в двоичном виде. Поменять местами третий бит с пятым. Результат умножить на 8 и проинвертировать.

8.	$y = \frac{(22 * 3 - 45 / 6) / 14}{(134 - 7 * 5) / 13} * 2 + 4$	Дано число в двоичном виде. Логически перемножить его с числом 28. Проинвертировать результат и умножить на 4. В полученном значении 4,5,6 разряды заменить на противоположные.
9.	$y = \frac{(12 - 15 / 2)}{15 / 2} + 12 / 7 - \frac{13 * 2}{5}$	Даны два числа в двоичном виде. Логически их перемножить и в результирующем значении поменять местами 7-ой разряд с 1-ым, 5-ый со 2-ым.
10.	$y = \frac{5 * 3 + 13 / 4}{1 + 12 / 5} - 28 + \frac{13}{3}$	Дано число в двоичном виде. Поменять местами четные разряды с нечетным. Результат проинвертировать и умножить на 4.
11.	$y = \frac{12 - 9}{(16 - 9) / 6} - \frac{3 * 2}{2 + 3} * (12 / 5)$	Дано число в двоичном виде. Вывернуть число «наизнанку» (разряды стоящие в середине сделать крайними). Результат разделить на 16.
12.	$y = \frac{14 * 3}{16 / 5 - 2 * 3} - \frac{14 / 4}{5 - 3}$	Даны два числа в двоичном виде. Из первого числа взять четыре младших разряда и поменять местами с четырьмя старшими разрядами второго числа. Результаты логически сложить и разделить на 8.
13.	$y = \frac{12 / 3 + 5*" + 16 / (8 - 5)}{15 / 2}$	Даны два числа в двоичном виде. Поменять местами 7,6,5,1- разряды первого числа с 0,2,3,4 разрядами второго числа соответственно. Результаты логически сложить и умножить на 8.
14.	$y = \frac{36 + (14 - 5) * 3}{(38 - 4 * 3) / 3} * 3 - 4$	Дано число в двоичном виде. Все нечетные разряды числа обнулить, а четные заменить на противоположные. Результат разделить на 4 и проинвертировать.
15.	$y = \frac{34 + (15 - 7) * 2}{36 / 4 + 35 / 8} - 34 * 2$	Даны два числа в двоичном виде (первое число размером в байт , второе число размером в слово). Первое число умножить на 16 и в полученном значении обнулить 3,5 разряды. Результат сложить со старшей частью второго числа.
16.	$y = \frac{(123 - 4) / 5 + (6 - 3) / 4}{36 + 4 * 3} - 34$	Даны два числа в двоичном виде. Обнулить в первом числе 3,5,6 разряды и разделить полученное значение на 8, второе число умножить

		на 2 и логически сложить с первым. Результат проинвертировать.
17.	$y = \frac{34 + (15 - 7) * 2}{36 / 4 + 35 / 8} - 34 * 2$	Дано двоичное число. В старшей части числа все четные биты заменить на противоположные. В младшей части числа все нечетные биты обнулить. Результат разделить на 16.
18.	$y = (24 - 14) * 36 + \frac{114 - 7 * 3}{136 / 5 + 7 * 3}$	Даны два числа в двоичном виде. В первом числе поменять местами старшую и младшую части числа. Во втором 1-ый и 4-ый разряды поменять местами с 3,7-мым разрядами соответственно. Результаты логически сложить и умножить на 4.
19.	$y = 37 - 14 * 3 * \frac{2 * (26 + 3) - 44 * 6}{48 - 14 / 2}$	Дано число в двоичном виде. Разделить его на две составляющие: в первую войдут только четные разряды, во вторую только нечетные разряды. Их логически перемножить и результат умножить на 16.
20.	$y = \frac{(258 - 140) / (34 - 12)}{36 + 12 / 5} * (36 + 3) - 17 / 4$	Даны два числа в двоичном виде. Первое число умножить на 4. второе разделить на 2 . Результаты логически сложить. 0-ой и 7-ой разряды, в полученном значении, поменять местами
21.	$y = \frac{-15 / 6 + (34 - 7 * 2) * 2}{16 + 13 / 8} - 14$	Дано число в двоичном виде. Поменять местами значения четных и нечетных разрядов. Полученное число проинвертировать и умножить на 8.
22.	$y = \frac{\frac{37 + 14 * 5}{2} + 5 * 6}{17 - 4 / 2} + 5 * 4$	Даны четыре числа в двоичном виде. Составить пятое число, которое состоит из 0-го и 1-го битов первого числа, 2-го и 3-го битов второго числа, 4,5-ые биты из третьего числа, 6,7-ой биты из четвертого числа. Полученное значение проинвертировать и разделить на 16.
23.	$y = 17 * 4 + \frac{5 - 8 / 3}{\frac{14 + 5}{4} + \frac{16 * 6 + 5}{10}}$	Дано двоичное число. Поменять местами 3-ий разряд с 7-ым. Полученное значение разделить на 8 и логически сложить с числом 56.

24.	$y = 35 * 4 + \dfrac{\dfrac{17 - 8/3}{2} + 4 * 8}{3 + 2}$	Дано число в двоичном виде. 0,1,4,5-ые разряды заменить на противоположные. Остальные занести в отдельный регистр, поставив их на 0,1,4,5-ые биты соответственно. Полученные значения логически перемножить
25.	$y = 3 * 6 + 4 * 4 / 3 * (48 + 16 / 2) * \dfrac{5 + 8}{14 - 3}$	Даны два числа в двоичном виде. Поменять местами четные разряды одного числа с нечетными разрядами другого числа. Полученные значения логически перемножить и все разряды числа заменить на противоположные.
26.	$y = \dfrac{14 - 4}{2} * (5 + 16 / 4) * \dfrac{17 + 8/3}{\dfrac{8}{4} + 2 * 5}$	Дано число в двоичном виде. Записать его наоборот. Полученное значение умножить на 8 и логически сложить с числом 17.
27.	$y = \dfrac{171 + 4 * 3 * 2}{\dfrac{36 - 4}{3} - 17} + (5 + 6) * 8 * \dfrac{17}{5 + 2}$	Дано число в двоичном виде. Обнулить 2,3,4 разряды тремя способами.
28.	$y = \dfrac{250 - 4 * 6}{\dfrac{33 + 4}{2} - 17} * (2 + 4) - 17$	Дано число в двоичном виде. Заменить 0,1,3-ие разряды на противоположные. Все остальные разряды сделать единичными, результат разделить на 16.
29.	$y = 7 + 4 * 6 + 5 / 2 * 13 + \dfrac{\dfrac{14 - 4}{2} + 17 * 6}{\dfrac{36 - 4}{3} + 2}$	Даны два числа в двоичном виде. Поменять местами четные и нечетные разряды двумя способами.
30.	$y = \dfrac{3 * 4 + 5 / 7}{5 - (3 - 4) / 2} - \dfrac{17 + 2}{3 * 6 + 6}$	Дано число в двоичном виде. Все четные разряды сделать единицами. Полученные значения разделить на 8 и поменять местами правую и левую части.

31.	$y = \frac{25 / 4 + 5 * 6}{36} + \frac{37}{\frac{8 * 6 + 5}{3} + 3 * 2}$	Даны два числа в двоичном виде. 1,3,7-ой разряды первого числа логически перемножить с 1,3,7-ым разрядом второго числа, результат разделить на 8 и проинвертировать.
32.	$y = \frac{37 / 2 + \frac{4 * 6 + 5}{4}}{15 - 7} * (5 + 8) - 136$	Даны два числа в двоичном виде. Все четные разряды первого числа логически сложить с четными разрядами второго числа, а нечетные обнулить. Результат разделить на 4 и проинвертировать.
33.	$y = \frac{3 * 4 + 5 * 6}{7 / 2 + 4 / 3 + 12 / 3} * \frac{16}{2} 3 * 6$	Дано число в двоичном виде. К старшей части числа логически прибавить проинвертированную младшую части числа. Результат разделить на 2.
34.	$y = \frac{49 / 7 + 42 / 6 + 6 * 2}{8 * 5 / 9 - 4 * 6} + \frac{23 + 12}{26 - 4} * 5$	Дано число в двоичном виде. К четным битам числа логически прибавить нечетные биты числа. Результат разделить на 8.
35.	$y = \frac{\frac{36 + 4 * 5}{2 + 6} - 14 / 7 + 1}{\frac{36 - 4}{2} + 12} - 7 * 6 * \frac{3 + 4}{12 - 7}$	Дано число в двоичном виде. Сделать 2,5,7-ой разряды единичными тремя способами.
36.	$y = \frac{\frac{2 + 3}{2} + \frac{4 * 6 + 25 * 5}{36 - 4}}{\frac{250 * 2 - 14}{37}} + (12 - 4) * (37 - 8)$	Даны два числа в двоичном виде. Поменять местами четные разряды первого числа с нечетными разрядами второго числа. Первое умножить на 2, а второе разделить на 2, результаты логически сложить.
37.	$y = \frac{\frac{320}{4} + \frac{256}{2} - 14}{56 * 2} * (3 + 2)$	Даны два числа в двоичном виде. В первом числе 2,4,6,7-ой разряды заменить на противоположные. Второе число разделить на 8. результаты логически сложить и проинвертировать.

38.	$y = \frac{(12 + 4 - 7) * 5}{36 * 4 - \frac{17}{2}} * \frac{(2 + 5)}{7} + 14$	Даны три числа в двоичном виде. Старшую часть первого числа логически сложить с младшей частью третьего числа, а младшую часть первого числа логически умножить на младшую часть второго числа. Результат разделить на 4.
39.	$y = \frac{34 + 260 / 2 + 170 * 3}{\frac{36 + 8}{2} + 5 * 6} - \frac{34}{2} * (6 + 7)$	Даны два числа в двоичном виде. Получить третье число путем логического сложения четных разрядов первого числа и нечетных разрядов второго числа. Получить четвертое число путем логического умножения нечетных разрядов первого числа и четных разрядов второго числа. Третье и четвертое числа проинвертировать и разделить на 4.
40.	$y = \frac{15 + 30 / 2 + 3 * 5}{\frac{36 + 4}{2} + 20 - 3}$	Даны два числа в двоичном виде. Первое число проинвертировать и разделить на 2. Второе число умножить на 4. Результаты логически сложить и первые четыре разряда заменить на противоположные.

2 АРИФМЕТИЧЕСКИЕ КОМАНДЫ И КОМАНДЫ ПЕРЕХОДОВ

2.1 Получение символов с клавиатуры

Ввод информации с клавиатуры - один из основных способов взаимодействия с компьютером IBM PC. DOS обеспечивает ряд функций, с помощью которых программа на ассемблере может обрабатывать нажатия клавиш.

Возможно, одним из наиболее простых способов получения символов клавиш является функция "Ввод с клавиатуры", то есть функция DOS номер 1. Функции DOS вызываются путем помещения номера функции в регистр AH и выполнения затем инструкции INT 21h. (Действительная работа инструкции INT несколько более сложна, но сейчас вам требуется только знать, что каждый раз при вызове функции DOS вы должны выполнять инструкцию INT 21h.) Следующий набранный на клавиатуре символ возвращается в регистре AL.

Например, когда выполняется код:

```
mov ah,1
int 21h
```

операционная система DOS помещает следующий набранный на клавиатуре символ в регистр AL. Заметим, что если клавиша не нажата, DOS будет ждать, когда она будет нажата; поэтому для выполнения данной функции может потребоваться неопределенное время.

2.2 Вывод символов на экран

Если нажатия клавиш означают взаимодействие пользователя с программным обеспечением, то экран является дополнением. IBM PC оснащаются дисплеями различных типов, начиная от цветного текстового до графического с высоким разрешением, но в данный момент мы рассмотрим только вывод символов.

Функция DOS с номером 2 обеспечивает наиболее непосредственный путь вывода символа на экран. Для этого нужно просто поместить 2 в регистр AH и выводимый символ в регистр DL, а затем вызвать DOS с помощью INT 21h. Следующий код отображает каждый введенный символ на экране:

```
mov ah,1
int 21h ; получить код следующей нажатой клавиши
mov ah,2
mov dl,al ; переместить считанный символ из AL в DL
int 21h ; вывести его на экран
```

Есть еще одно замечание, которое нужно сделать относительно клавиатуры, экрана и файлового ввода и вывода на языке Ассемблера. Те из вас, кто пользовался функциями scanf и printf в языке Си или функциями Readln n Writeln в Паскале, возможно с удивлением узнают, что в DOS не предусмотрено форматного ввода и вывода. DOS выполняет только посимвольный или построчный ввод-вывод. В Си для печати целой переменной вам требуется сделать следующее:

```
printf("\\d\n",i);
```

Си автоматически преобразует целое значение, которое хранится в 16-битовой ячейке памяти, в строку символов кода ASCII и печатает символы. В Ассемблере ваша программа должна явно преобразовывать переменные в строки символов, перед тем, как вывести их на экран. Аналогично, DOS знает только, как считывать символы и строки символов с клавиатуры, поэтому вам придется писать программы, преобразующие вводимые пользователем строки и символы в другие данные.

2.3 Безусловные переходы

Основной инструкцией перехода в наборе инструкций процессора 8086 является инструкция JMP. Эта инструкция указывает процессору 8086, что в качестве следующей за JMP инструкцией нужно выполнить инструкцию по целевой метке. Например, после завершения выполнения фрагмента программы:

```
mov ax,1
jmp AddTwoToAX
AddOneToAx:
 inc ax
 jmp AXIsSet
AddTwoToAX:
 inc ax
AXIsSet:
```

регистр AX будет содержать значение 3, а инструкции ADD и JMP, следующие за меткой AddOneToAX, никогда выполнены не будут. Здесь инструкция:

```
jmp AddTwoToAX
```

указывает процессору 8086, что нужно установить указатель инструкций IP в значение смещения метки AddTwoToAX; поэтому следующей выполняемой инструкцией будет инструкция:

```
add ax,2
```

Иногда совместно с инструкцией JMP используется операция SHORT. Для указания на целевую метку инструкция JMP обычно использует 16-битовое смещение. Операция SHORT указывает Турбо Ассемблеру, что нужно использовать не 16-битовое, а 8-битовое смещение (что позволяет сэкономить в инструкции JMP один байт). Например, последний фрагмент программы можно переписать так, что он станет на два байта короче:

```
mov ax,1
 jmp SHORT AddTwoToAX
AddOneToAx:
 inc ax
 jmp SHORT AXIsSet
AddTwoToAX:
 inc ax
AXIsSet:
```

Недостаток использования операции SHORT (короткий) состоит в том, что короткие переходы могут осуществлять передачу управления на метки, отстоящие от инструкции JMP не далее, чем на 128 байтов, поэтому в некоторых случаях Турбо Ассемблер может сообщать вам, что метка недостижима с помощью короткого перехода. К тому же операцию SHORT имеет смысл использовать для ссылок вперед, поскольку для переходов назад (на предшествующие метки) Турбо Ассемблер автоматически использует короткие переходы, если на метку можно перейти с помощью короткого перехода, и длинные в противном случае.

2.4 Условные переходы

Описанные в предыдущем разделе инструкции переходов - это только часть того, что вам потребуется для написания полезных программ. В действительности необходима возможность писать такие программы, которые могут принимать решения. Именно это можно делать с помощью операций условных переходов.

Инструкция условного перехода может осуществлять или нет переход на целевую (указанную в ней) метку, в зависимости от состояния регистра флагов. Рассмотрим следующий пример:

```
mov ah,1 ;функция DOS ввода с клавиатуры
 int 21h ; получить следующую нажатую клавишу
 cmp al,'A' ; была нажата буква "A"?
 je AWasTyped ; да, обработать ее
 mov [TampByte],al ; нет, сохранить символ
 .
 .
 .
AWasTyped:
 push ax ; сохранить символ в стеке
```

Сначала в данной программе с помощью функции операционной системы DOS воспринимается нажатая клавиша. Затем для сравнения введенного символа с символом A используется инструкция CMP. Эта инструкция аналогична инструкции SUB, только ее выполнение ни на что не влияет, поскольку назначение данной инструкции состоит в том, чтобы можно было сравнить два операнда, установив флаги так же, как это делается в инструкции SUB. Поэтому в предыдущем примере флаг нуля устанавливается в значение 1 только в том случае, если регистр AL содержит символ A.

Теперь мы подошли к основному моменту. Инструкция JE представляет инструкцию условного перехода, которая осуществляет передачу управления только в том случае, если флаг нуля равен 1. В противном случае выполняется инструкция, непосредственно следующая за инструкцией JE (в данном случае - инструкция MOV). Флаг нуля в данном примере будет установлен только в случае нажатия

клавиши A; и только в этом случае процессор 8086 перейдет к выполнению инструкции с меткой AWasTyped, то есть инструкции PUSH.

Набор инструкций процессора 8086 предусматривает большое разнообразие инструкций условных переходов, что позволяет вам осуществлять переход почти по любому флагу или их комбинации. Можно осуществлять условный переход по состоянию нуля, переноса, по знаку, четности или флагу переполнения и по комбинации флагов, показывающих результаты операций чисел со знаками.

Перечень инструкций условных переходов приводится в таблице 1.

Таблица 1 - Инструкции условных переходов

Название	**Значение**	**Проверяемые флаги**
JB/JNAE	Перейти, если меньше / перейти, если не больше или равно	CF = 1
JAE/JNB	Перейти, если больше или равно / перейти, если не меньше	CF = 0
JBE/JNA	Перейти, если меньше или равно / перейти, если не больше	CF = 1 или ZF = 1
JA/JNBE	Перейти, если больше / перейти, если не меньше или равно	CF = 0 и ZF = 0
JE/JZ	Перейти, если равно	ZF = 1
JNE/JNZ	Перейти, если не равно	ZF = 0
JL/JNGE	Перейти, если меньше чем / перейти, если не больше чем или равно	SF = OF
JGE/JNL	Перейти, если больше чем или равно /перейти, если не меньше чем	SF = OF
JLE/JNLE	Перейти, если меньше чем или равно / перейти, если не больше, чем	ZF = 1 или SF = OF

JG/JNLE	Перейти, если больше чем / перейти, если не меньше чем или равно	ZF = 0 или SF = OF
JP/JPE	Перейти по четности	PF = 1
JNP/JPO	Перейти по нечетности	PF = 0
JS	Перейти по знаку	SF = 1
JNS	Перейти, если знак не установлен	SF = 0
JC	Перейти при наличии переноса	CF = 1
JNC	Перейти при отсутствии переноса	CF = 0
JO	Перейти по переполнению	OF = 1
JNO	Перейти при отсутствии переполнения	OF = 0

CF - флаг переноса, SF - флаг знака, OF - флаг переполнения, ZF - флаг нуля, PF - флаг четности.

Не смотря на свою гибкость, инструкции условного перехода имеют также серьезные ограничения, поскольку переходы в них всегда короткие. Другими словами, целевая метка, указанная в инструкции условного перехода, должна отстоять от инструкции перехода не более, чем на 128 байт. Например, Турбо Ассемблер не может ассемблировать:

```
JumpTarget:
.
.
.
DB 1000 DUP (?)
.
.
.
dec ax
jnz JumpTarget
```

так как метка JumpTarget отстоит от инструкции JNZ более чем на 1000 байт. В данном случае нужно сделать следующее:

```
JumpTarget:
.
.
.
```

```
DB 1000 DUP (?)
.
.
.
dec ax
jnz SkipJump
jmp JumpTarget
SkipJump:
```

где условный переход применяется для того, чтобы определить, нужно ли выполнить длинный безусловные переход.

Командам условного перехода может предшествовать любая команда, изменяющая состояние флагов.

2.5 Пример

Рассмотрим простейший пример с использованием ввода, вывода и различных видов переходов.

Необходимо ввести с клавиатуры значение двух переменных a и x. Если a<x, то сложить их значения, а иначе из x отнять a.

```
data segment
a db ?
x db ?
per db 10,13,'$'
mesa db 10,13,'Input a: $'
mesx db 10,13,'Input x: $',10,13
data ends
s segment stack
db 128 dup(?)
s ends
code segment
main:
assume ss:s,ds:data,cs:code
mov ax,data
mov ds,ax
mov dx,offset mesa
mov ah,9 ;Приглашение на ввод a
int 21h

mov ah,1 ;Считывание нажатого символа
```

```
int 21h

mov a,al ;Запись считанного символа в a

mov dx,offset mesx
mov ah,9 ;Приглашение на ввод x
int 21h

mov ah,1 ;Считывание нажатого символа
int 21h
mov x,al ;Запись считанного символа в x

mov dx,offset per
mov ah,9 ;Перевод строки
int 21h

mov al,x
cmp a,al
jl Lower ;Если a<x,то перейти на метку Lower. Иначе на метку Higher
Higher:
mov al,a
sub al,x
add al,30h ;Коррекция по вычитанию
jmp short l1
lower:
mov al,x ;В регистр al записываем результат сложения a и x
add al,a
sub al,30h ;Корекция по сложению
l1:
mov dl,al
mov ah,2 ;Вывод содержимого dl на экран
int 21h

mov ah,0 ;Ожидание нажатия клавиши
int 16h

mov ah,4ch
int 21h
code ends
end main
```

Остановимся подробнее на строках, в которых происходит коррекция по сложению и вычитанию. Так как Ассемблер не способен обрабатывать просто

десятичные числа, то необходимо придумывать алгоритмы обработки самостоятельно. Удобнее всего приводить их к нераспакованному десятичному виду.

При чтении с клавиатуры происходит чтение именно символа, и в регистр записывается его код. Например у чисел 1 и 5 коды соответственно 31h и 35h. Чтобы привести к нераспакованному десятичному виду необходимо привести их к виду 01h и 05h. Эти коды имеют символы отличные от 1 и 5, но над ними гораздо удобнее выполнять арифметические операции. Существуют специальные команды коррекции, но о них вы узнаете позже.

2.6 Задания

В соответствии с вариантом напишите программу на языке ассемблера с полным описанием сегментов для вычисления значения y. Используйте, где требуется, 32х-разрядные регистры.

1	$y = y1 + y2;\ y1 = \begin{cases} a + x, & \text{если } x > a \\ 2a - x, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a * x, & \text{если } x > 10 \\ x, & \text{если } x <= 10 \end{cases}.$
2	$y = y1 - y2;\ y1 = \begin{cases} x - 2, & \text{если } x >= 2 \\ 8, & \text{если } x < 2 \end{cases};\ y2 = \begin{cases} 4, & \text{если } x = 0 \\ a - x, & \text{если } x <> 0 \end{cases}.$
3	$y = y1 * y2;\ y1 = \begin{cases} x - a, & \text{если } x > a \\ 5, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a, & \text{если a} > x \\ a * x, & \text{если a} <= x \end{cases}.$
4	$y = y1 + y2;\ y1 = \begin{cases} 2 - x, & \text{если } x < 2 \\ a + 3, & \text{если } x >= 2 \end{cases};\ y2 = \begin{cases} a - 1, & \text{если } x < a \\ a * x - 1, & \text{если } x >= a \end{cases}.$
5	$y = y1 - y2;\ y1 = \begin{cases} \lvert x \rvert, & \text{если } x < 0 \\ x - a, & \text{если } x >= 0 \end{cases};\ y2 = \begin{cases} a + x, & \text{если } x \bmod 3 = 1 \\ 7, & \text{в остальных случаях} \end{cases}.$
6	$y = y1 + y2;\ y1 = \begin{cases} x \bmod 4, & \text{если } x > a \\ a, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a * x, & \text{если } x / a > 3 \\ x, & \text{если } x / a <= 3 \end{cases}.$
7	$y = y1 + y2;\ y1 = \begin{cases} 4 - x, & \text{если } \lvert x \rvert < 3 \\ a + x, & \text{в остальных случаях} \end{cases};\ y2 = \begin{cases} 2, & \text{если } x \text{ четное} \\ a + 2, & \text{в остальных случаях} \end{cases}.$
8	$y = y1 + y2;\ y1 = \begin{cases} 4 * x, & \text{если } x <= 4 \\ x - a, & \text{если } x > 4 \end{cases};\ y2 = \begin{cases} 7, & \text{если } x \text{ нечетное} \\ x / 2 + a, & \text{в остальных случаях} \end{cases}.$
9	$y = y1 * y2;\ y1 = \begin{cases} a * x, & \text{если } x \bmod 3 = 2 \\ 9, & \text{в остальных случаях} \end{cases};\ y2 = \begin{cases} a - x, & \text{если } a > x \\ a + 2, & \text{если } a <= x \end{cases}.$

10	$y = y1 - y2;\ y1 = \begin{cases} a + \lvert x \rvert, & \text{если } x > a \\ a - 7, & \text{если } x <= a \end{cases};\ y2 = \begin{cases} a * 3, & \text{если } a > 3 \\ 11, & \text{если a <= 3} \end{cases}.$
11	$y = y1 \bmod y2;\ y1 = \begin{cases} 10 + x, & \text{если } x > 1 \\ \lvert x \rvert + a, & \text{если } x <= 1 \end{cases};\ y2 = \begin{cases} 2, & \text{если } x > 4 \\ x, & \text{если } x <= 4 \end{cases}.$
12	$y = y1 / y2;\ y1 = \begin{cases} 15 + x, & \text{если } x > 7 \\ \lvert a \rvert + 9, & \text{если } x <= -7 \end{cases};\ y2 = \begin{cases} 3, & \text{если x > 2} \\ \lvert x \rvert - 5, & \text{если } x <= 2 \end{cases}.$
13	$y = y1 * y2;\ y1 = \begin{cases} 3 + x, & \text{если } x = a \\ a - x, & \text{если } x <> a \end{cases};\ y2 = \begin{cases} \lvert a \rvert, & \text{если } a < x \\ \lvert a \rvert - x, & \text{если ```a >= } x \end{cases}.$
14	$y = y1 - y2;\ y1 = \begin{cases} 2 * x + a, & \text{если } x > 2 \\ 2 * x + 1, & \text{если } x <= 2 \end{cases};\ y2 = \begin{cases} \lvert x \rvert + 1, & \text{если x > 0} \\ a - 1, & \text{если } x <= 0 \end{cases}.$
15	$y = y1 \bmod y2;\ y1 = \begin{cases} 8 + \lvert x \rvert, & \text{если } x < 1 \\ \lvert a \rvert * 2, & \text{если } x >= 1 \end{cases};\ y2 = \begin{cases} 3, & \text{если } x = a \\ a + 1, & \text{если } x <> a \end{cases}.$
16	$y = y1 + y2;\ y1 = \begin{cases} 4 + x, & \text{если } x <= 3 \\ a * x, & \text{если } x > 3 \end{cases};\ y2 = \begin{cases} \lvert a \rvert - 2, & \text{если x > a} \\ x, & \text{если } x <= a \end{cases}.$
17	$y = y1 - y2;\ y1 = \begin{cases} a + \lvert x \rvert, & \text{если } x < 0 \\ x - a, & \text{если } x >= 0 \end{cases};\ y2 = \begin{cases} 7, & \text{если x < 3} \\ a, & \text{если } x >= 3 \end{cases}.$
18	$y = y1 \bmod y2;\ y1 = \begin{cases} 7 + x, & \text{если } x < 3 \\ \lvert a \rvert + x, & \text{если } x >= 3 \end{cases};\ y2 = \begin{cases} 1, & \text{если } x > 5 \\ a + x, & \text{если } x <= 5 \end{cases}.$
19	$y = \lvert y1 \rvert + y2;\ y1 = \begin{cases} -5, & \text{если } x > 4 \\ x - a, & \text{если } x <= 4 \end{cases};\ y2 = \begin{cases} \lvert a \rvert, & \text{если x > a} \\ 9, & \text{если } x <= a \end{cases}.$
20	$y = y1 * y2;\ y1 = \begin{cases} 2 * x, & \text{если } x < 5 \\ \lvert a \rvert + x, & \text{если } x >= 5 \end{cases};$
21	$y = y1 + \lvert y2 \rvert;\ y1 = \begin{cases} 3, & \text{если } x \bmod 3 = 1 \\ x - a, & \text{в остальных случаях} \end{cases};\ y2 = \begin{cases} a / x, & \text{если } x <> 0 \\ 4, & \text{если x = 0} \end{cases}.$
22	$y = y1 - y2;\ y1 = \begin{cases} \lvert x \rvert + \lvert a \rvert, & \text{если } x <= 3 \\ x * a, & \text{если } x > 3 \end{cases};\ y2 = \begin{cases} 3, & \text{если } a = x \\ a - x, & \text{если } a <> x \end{cases}.$
23	$y = y1 + y2;\ y1 = \begin{cases} 2 * x, & \text{если } \lvert x \rvert > 4 \\ 4 + a, & \text{в остальных случаях} \end{cases};\ y2 = \begin{cases} 9, & \text{если x = 0} \\ a / x, & \text{если } x <> 0 \end{cases}.$
24	$y = y1 * y2;\ y1 = \begin{cases} x, & \text{если } x \bmod 4 <> 2 \\ a + x, & \text{в остальных случаях} \end{cases};\ y2 = \begin{cases} a - x, & \text{если } x < a \\ a * x, & \text{если x >= } a \end{cases}.$

25	$y = y1 / y2 \; ; \; y1 = \begin{cases} 12, & \text{если} \quad x < 12 \\ x + 1, & \text{если} \quad x >= 12 \end{cases} ; \; y2 = \begin{cases} 2, & \text{если} \quad x > 2 \\ a + x, & \text{если} \quad x <= 2 \end{cases}.$

3 ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕР ЗАДАЧ С ИСПОЛЬЗОВАНИЕМ МАССИВОВ СТРОКОВЫХ ДАННЫХ

3.1 Предопределенный идентификатор $

В Турбо Ассемблере имеется несколько предопределенных идентификаторов (например, @data). Еще один простой, но удивительно полезный предопределенный идентификатор - это идентификатор $, который всегда установлен в текущее значение счетчика адреса. Другими словами, идентификатор $ всегда равен текущему смещению в сегменте, в котором Турбо Ассемблер в данный момент выполняет ассемблирование. $ представляет собой постоянное значение смещения, аналогичное OFFSET MemVar. Это позволяет использовать $ в выражениях или в любом месте, где допускается использование константы.

Идентификатор $ очень удобно использовать для вычисления длины данных и кода. Предположим, например, что вы хотите приравнять идентификатор STRING_LENGTH к длине строки в байтах. Без предопределенного идентификатора $ вам придется сделать следующее:

```
StringStart LABEL BYTE
 db 0dh,0ah,'Текстовая строка'odh,0ah
 StringEnd LABEL BYTE
 STRING_LENGTH EQU (StringEnd-StringStart)
```

а с помощью идентификатора $ вы можете записать:

```
StringStart LABEL BYTE
 db 0dh,0ah,'Текстовая строка'odh,0ah
 STRING_LENGTH EQU ($-StringStart)
```

Длину (в словах) массива слов можно вычислить следующим образом:

```
WordArray DW 90h, 25h, 0, 16h, 23h
 WORD_ARRAY_LENGTH EQU (($-WordArray)/2)
```

Вы, конечно, можете сосчитать отдельные элементы вручную, но для больших массивов и строк это довольно затруднительно.

3.2 Цепочные команды

3.2.1 Инструкция LODS

Инструкция LODS, которая загружает байт или слово из памяти в аккумулятор (накопитель), подразделяется на две инструкции - LODSB и LODSW. Инструкция LODSB загружает байт, адресуемый с помощью пары регистров DS:SI, в регистр AL и уменьшает или увеличивает регистр SI (в зависимости от состояния флага направления). Если флаг направления равен 0 (установлен с помощью инструкции CLD), то регистр SI увеличивается, а если флаг направления равен 1 (установлен с помощью инструкции STD), то регистр SI уменьшается. И это верно не только для инструкции LODSB; флаг направления управляет направлением, в котором изменяются все регистры-указатели строковых инструкций.

Например, в следующем фрагменте программы:

```
cld
mov si,0
lodsb
```

инструкция LODSB загружает регистр AL содержимым байта со смещением 0 в сегменте данных и увеличивает значение регистра SI на 1.Это эквивалентно выполнению следующих инструкций:

```
mov si,0
mov al,[si]
inc si
```

Однако инструкция LODSB работает существенно быстрее (и занимает на два байта меньше), чем инструкции:

```
mov al,[si]
inc si
```

Инструкция LODSW аналогична инструкции LODSB. Она сохраняет в регистре AX слово, адресуемое парой регистров DS:SI; а значение регистра SI уменьшается или увеличивается на 2, а не на 1. Например, инструкции:

```
std
mov si,0
lodsw
```

загружают слово со смещением 10 в сегменте данных в регистр RU, а затем значение SI уменьшается на 2.

3.2.2 Инструкция STOS

Инструкция STOS - это дополнение инструкции LODS. Она записывает значение размером в байт или слово из аккумулятора в ячейку памяти, на которую указывает пара регистров ES:DI, а затем увеличивает или уменьшает DI. Инструкция STOSB записывает байт, содержащийся в регистре AL, в ячейку памяти по адресу ES:DI, а затем увеличивает или уменьшает регистр DI, в зависимости от флага направления. Например, инструкции:

```
std
mov di,0ffffh
mov al,55h
stosb
```

записывают значение 55h в байт со смещением 0FFFFh в сегменте, на который указывает регистр ES, а затем уменьшает DI до значения 0FFFEh.

Инструкция STOSW работает аналогично, записывая значение размером в слово, содержащееся в регистре AX, по адресу ES:DI, а затем увеличивает или уменьшает значение регистра DI на 2. Например, инструкции:

```
cld
mov di,0ffeh
mov al,102h
stosw
```

записывают значение 102h размером в слово, записанное в регистре AX, по смещению 0FFEh в сегменте, на который указывает регистр ES, а затем значение регистра DI увеличивается до 1000h.

Инструкции LODS и STOS можно прекрасно использовать вместе для копирования буферов. Например, следующая подпрограмма копирует завершающуюся нулевым символом строку, записанную по адресу DS:SI, в строку по адресу ES:DI:

```
;
; Подпрограмма для копирования завершающейся нулем строки
; в другую строку
;
; Ввод:
; DS:SI - строка, из которой выполняется копирование
; ES:DI - строка, в которую выполняется копирование
;
; Вывод: нет
;
; Изменяемые регистры: AL, SI, DI
;
CopyString PROC
 cld ; обеспечить увеличение SI и
 ; DI в строковых инструкциях
CopyStringLoop:
 lodsb ; получить символ исходной
 ; строки
 stosb ; записать символ в выходную
 ; строку
 cmp al,0 ; последним символом строки
 ; был 0?
 jnz CopyStringLoop ; нет, обработать следующий символ
 ret ; да, выполнено
CopyString ENDP
```

Аналогично вы можете использовать инструкции LODS и STOS для копирования блока байт, которые не завершаются нулем, используя для этого цикл:

```
mov cx,ARRAY_LENGTH_IN_WORDS ; размер массива
mov si,OFFSET SourceArray ; исходный массив
```

```
mov ax,SEG SourceArray
mov dx,ax
mov di,OFFSET DestArray ; целевой массив
mov ax,SEG DestArray
mov es,ax
cld
CopyLoop:
lodsw
stosw
loop CopyLoop
```

Однако для перемещения байта или слова из одного места в памяти в другое есть еще более лучший способ. Это инструкция MOVS.

3.2.3 Инструкция MOVS

Инструкция MOVS аналогична инструкциям LODS и STOS, если их объединить в одну инструкцию. Эта инструкция считывает байт или слово, записанное по адресу DS:SI, а затем записывает это значение по адресу, определяемому парой регистров ES:DI. Слово или байт не передается при этом через регистры, поэтому содержимое регистра AX не изменяется. Инструкция MOVSB имеет минимально возможную для инструкции длину. Она занимает только один байт, а работает еще быстрее, чем комбинация инструкций LODS и STOS. С применением инструкции MOVS последний пример приобретает вид:

```
mov cx,ARRAY_LENGTH_IN_WORDS
mov si,OFFSET SourceArray
mov ax,SEG SourceArray
mov ds,ax
mov di,OFFSET DestArray
mov ax,SEG DestArray
mov es,ax
cld
CopyLoop:
movsw
loop CopyLoop
```

3.2.4 Повторение строковой инструкции

Хотя в последнем примере код выглядит довольно эффективным, неплохо было бы избавиться от инструкции LOOP и перемещать весь массив с помощью одной инструкции. Инструкции процессора 8086 предоставляют такую возможность. Это форма строковых инструкций с префиксом REP.

Префикс повторения REP - это не инструкция, а префикс инструкции. Префикс инструкции изменяет работу последующей инструкции. Префикс REP делает следующее: он указывает, что последующую инструкцию нужно повторно выполнять до тех пор, пока содержимое регистра CX не станет равным 0. (Если регистр CX равен 0 в начале выполнения инструкции, то инструкция выполняется 0 раз, другими словами, никаких действий не производится.)

Используя префикс REP, можно заменить в последнем примере инструкции:

```
CopyLoop:
movsw
loop CopyLoop
```

на инструкцию:

```
rep movsb
```

Эта инструкция будет перемещать блок из 65535 слов (0FFFFh) из памяти, начинающейся с адреса DS:SI, в память, начинающуюся с адреса, определяемого регистрами ES:DI.

Конечно, для выполнения инструкции 65535 раз потребуется гораздо больше времени, чем для выполнения инструкции один раз, ведь для обращения ко всей этой памяти требуется время. Однако каждое повторение (с помощью префикса) строковой инструкции выполняется быстрее, чем выполнение одной строковой инструкции. Это позволяет получить очень быстрый способ чтения из памяти, записи в память и копирования.

Префикс REP можно использовать не только с инструкцией MOVS, но также и с инструкциями LODS и STOS (и инструкциями SCAS и CMPS - это мы обсудим позднее). Инструкцию STOS можно с успехом повторять для очистки или заполнения блоков памяти, например:

```
cld
mov ax,SEG WordArray
mov es,ax
mov di,OFFSET WordArray
sub ax,ax
mov cx,WORD_ARRAY_LENGTH
rep stosw
```

Здесь массив WordArray заполняется нулями. Для повторения инструкции LODS соответствующее полезное приложение придумать трудно.

Префикс REP вызывает повторение только строковой инструкции. Инструкция типа:

```
rep mov ax,[bx]
```

не имеет смысла. В этом случае префикс REP игнорируется и выполняется инструкция:

```
mov ax,[bx]
```

3.2.4 Сравнение строк

Команда CMPS сравнивает содержимое одной области памяти (адресуемой регистрами DS:SI) с содержимым другой области (адресуемой как ES:DI). В зависимости от флага DF команда CMPS также увеличивает или уменьшает адреса в регистрах SI и DI на 1 для байта или на 2 для слова. Команда CMPS устанавливает флаги AF, CF, OF, PF, SF и ZF. При использовании префикса REP в регистре CX должна находиться длина сравниваемых полей. Команда CMPS может сравнивать любое число байт или слов. Рассмотрим процесс сравнения двух строк, содержащих имена JEAN и JOAN. Сравнение побайтно слева направо приводит к следующему:

```
J : J Равно
E : O Не равно (E меньше O)
A : A Равно
N : N Равно
```

Сравнение всех четырех байт заканчивается сравнением N:N - равно/нуль. Так как имена "не равны", операция должна прекратиться, как только будет обнаружено условие "не равно". Для этих целей команда REP имеет модификацию REPE, которая повторяет сравнение до тех пор, пока сравниваемые элементы равны, или регистр CX не равен нулю. Кодируется повторяющееся однобайтовое сравнение следующим образом:

```
REPE CMPSB
```

3.3 Режимы адресации к памяти

Как при использовании операнда в памяти задать ту ячейку памяти, с которой вы хотите работать? Очевидный ответ состоит в том, чтобы присвоить нужной переменной в памяти имя (как мы это делали в последнем разделе). С помощью, например, следующих операторов вы можете вычесть переменную памяти Debts (долги) из переменной памяти Assets (имущество):

```
Assets DW ?
Debts DW ?
.
.
.
mov ax,[Debts]
sub [Assets],ax
```

Однако адресация к памяти имеет и более глубокий смысл, который не бросается в глаза. Предположим, у вас имеется символьная строка с именем CharString, содержащая буквы ABCDEFGHIGKLM, которые начинаются в сегменте данных со смещения 100, как показано на Рис. 1. Каким образом можно считать

девятый символ (I), который расположен по адресу 108? В языке Си вы можете просто использовать оператор:

```
C = CharString[8];
```

(в Си элементы нумеруются с 0), а в Паскале:

```
C := CharString[9];
```

Как же это можно сделать в Ассемблере? Прямая ссылка на строку CharString здесь, конечно, не подходит, так как первым символом является символ A.

```
 99 | ? |
 |--------------|
 CharString ---> 100 | 'A' |
 |--------------|
 101 | 'B' |
 |--------------|
 102 | 'C' |
 |--------------|
 103 | 'D' |
 |--------------|
 104 | 'E' |
 |--------------|
 105 | 'F' |
 |--------------|
 106 | 'G' |
 |--------------|
 107 | 'H' |
 |--------------|
 108 | 'I' |
```

|----------|

109 | 'J' |

|----------|

110 | 'K' |

|----------|

111 | 'L' |

|----------|

112 | 'M' |

|----------|

113 | 0 |

|----------|

114 | ? |

|----------|

. .

. .

Рисунок 1 – Ячейки памяти со строкой символов CharString.

В действительности язык Ассемблера обеспечивает несколько различных способов адресации к строкам символов, массивам и буферам данных. Наиболее простой способ состоит в том, чтобы считать девятый по счету символ строки CharString:

```
.DATA
CharString DB 'ABCDEFGHIJKLM'
.
.
.
.CODE
.
.
.
mov ax,@Data
mov ds,ax
```

```
mov al,[CharString+8]
```

В данном случае это то же самое, что:

```
mov al,[100+8]
```

так как CharString начинается со смещения 100. Все, что заключено в квадратные скобки, интерпретируется Турбо Ассемблером, как адрес; поэтому смещение CharString и 8 складывается и используется в качестве адреса памяти. Инструкция принимает вид:

```
mov al,[108]
```

как показано на Рис. 2.

```
. .
. .
| |
|--------------|
99 | ? |
|--------------|
CharString ------> 100 | 'A' |
|--------------|
101 | 'B' |
|--------------|
102 | 'C' |
|--------------|
103 | 'D' |
|--------------|
104 | 'E' |
|--------------|
105 | 'F' |
```

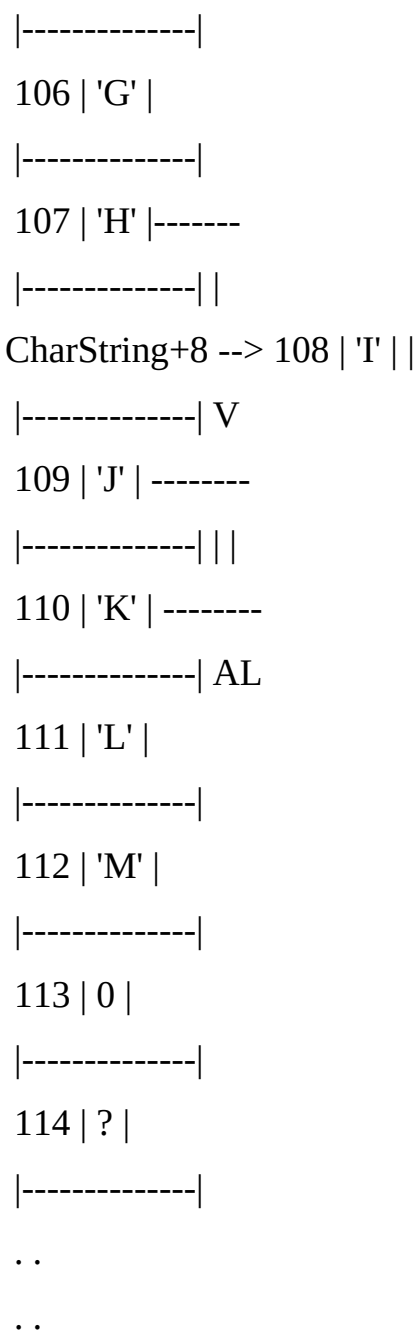


Рисунок 2 – Адресация строки символов строки CharString.

Такой тип адресации, когда ячейка памяти задается ее именем, плюс некоторая константа, называется непосредственной (прямой) адресацией. Хотя непосредственная адресация - это хороший метод, она не отличается достаточной гибкостью, поскольку обращение выполняется каждый раз по одному и тому же адресу памяти. Поэтому давайте рассмотрим другой, более гибкий путь адресации памяти.

Рассмотрим следующий фрагмент программы, где в регистр AL также загружается девятый символ CharString:

```
mov bx,OFFSET CharString+8
mov al,[bx]
```

В данном примере для ссылки на девятый символ используется регистр BX. Первая инструкция загружает в регистр BX смещение CharString (вспомните о том, что операция OFFSET возвращает смещение метки в памяти), плюс 8. (Вычисление OFFSET и сложение для этого выражения выполняется Турбо Ассемблером во время ассемблирования.) Вторая инструкция определяет, что AL нужно сложить с содержимым по смещению в памяти, на которое указывает регистр BX (см. Рис. 3).

```
mov al,[108]
```

как показано на рисунке 3.

```
. .
. .
| |
|--------------|
 99 | ? |
|--------------|
CharString ------> 100 | 'A' |
|--------------|
 101 | 'B' |
|--------------|
 102 | 'C' |
|--------------|
 103 | 'D' |
|--------------|
```

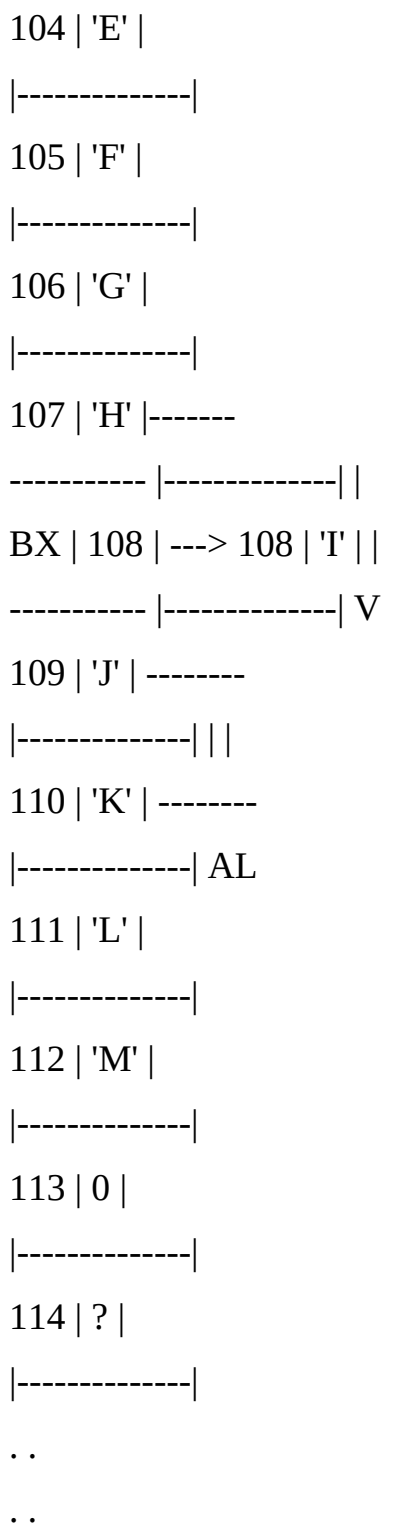


Рисунок 3 – Использование регистра BX для адресации к строке CharString.

Квадратные скобки показывают, что в качестве операнда-источника должна быть использоваться ячейка, на которую указывает регистр BX, а не сам регистр BX.

Не забывайте указывать квадратные скобки при использовании BX в качестве указателя памяти. Например:

```
mov ax,[bx] ; загрузить AX из ячейки памяти,
; на которую указывает BX
```

и

```
mov ax,bx ; загрузить в AX содержимое
; регистра BX
```

это две совершенно различные инструкции.

Может возникнуть вопрос, зачем сначала загружать в регистр BX смещение ячейки памяти и затем использовать BX, как указатель, если тоже самое можно сделать с помощью одной инструкции с непосредственным операндом? Особое свойство регистров, используемых в качестве указателей, состоит в том, что в отличие от инструкций, использующих непосредственные операнды, инструкции, использующие в качестве указателей регистры, могут ссылаться в разное время (в процессе выполнения программы) на разные ячейки памяти.

Предположим, вы хотите определить последний символ завершающейся нулем строки CharString. аЧтобы это сделать, вы должны, начиная с первого символа строки CharString, найти завершающий строку нулевой байт, затем вернуться назад на один символ и считать этот последний символ. Это невозможно сделать с помощью непосредственной адресации, так как строка может иметь произвольную длину. Использование регистра BX значительно облегчает задачу:

```
mov bx,OFFSET CharString ; указывает на строку
FindLastCharLoop:
mov al,[bx] ; получить следующий
; символ строки
cmp al,0 ; это нулевой байт?
je FoundEndOfString ; да, вернуться на
; один символ
inc bx
```

```
jmp FilnLastCharLoop ; проверить следующий
; символ
FoundEndOfString:
dec bx ; снова указывает на
; последний символ
mov al,[bx] ; получить последний
. ; символ строки
```

Если вы собираетесь выполнять в памяти поиск символов или слов, работать с массивами, или копировать блоки данных, вы поймете, что использование регистров-указателей дает неоценимую помощь.

BX - это не единственный регистр, который можно использовать для ссылки на память. Допускается также использовать вместе с необязательным значением-константой или меткой регистры BP, SI и DI. Общий вид операндов в памяти выглядит следующим образом:

```
[базовый регистр + индексный регистр + смещение]
```

где базовый регистр - это BX или BP, индексный регистр - SI или DI, а смещение - любая 16-битовая константа, включая метки и выражения. Каждый раз, когда выполняется инструкция, использующая операнд в памяти, процессором 8086 эти три компоненты складываются. Каждая из трех частей операнда в памяти является необязательной, хотя (это очевидно) вы должны использовать один из трех элементов, иначе вы не получите адреса в памяти. Вот как элементы операнда в памяти выглядят в другом формате:

```
BX SI
или + или + Смещение
BP DI
(база) (индекс)
```

Существует 16 способов задания адреса в памяти:

```
[смещение] [bp+смещение]
[bx] [bx+смещение]
```

```
[si]    [si+смещение]
[di]    [di+смещение]
[bx+si]         [bx+si+смещение]
[bx+di [bx+di+смещение]
[bp+si] [bp+si+смещение]
[bp+di] [bp+di+смещение]
```

где смещение - это то, что можно свести к 16-битовому постоянному значению.

Поведение регистра BP несколько отличается от регистра BX. Вспомните, что в то время как регистр BX используется, как смещение внутри сегмента данных, регистр BP используется, как смещение в сегменте стека. Это означает, что регистр BP не может обычно использоваться для адресации к строке CharString, которая находится в сегменте данных. . В данный момент достаточно знать, что регистр BP можно использовать так же, как мы использовали в примерах регистр BX, только адресуемые данные должны в этом случае находиться в стеке.

(На самом деле регистр BP можно использовать и для адресации к сегменту данных, а BX, SI и DI - для адресации к сегменту стека, дополнительному сегменту или сегменту кода. Для этого используются префиксы переопределения сегментов (segment override prefixes). О некоторых из них мы расскажем в Главе 10. Однако в большинстве случаев они вам не понадобятся, поэтому пока мы просто забудем об их существовании.)

Наконец, квадратные скобки, в которые заключаются непосредственные адреса, являются необязательными. То есть инструкции:

```
mov al,[MemVar]
```

и

```
mov al,MemVar
```

выполняют одни и те же действия. Тем не менее мы настоятельно рекомендуем вам заключать все ссылки на память в квадратные скобки. Это поможет избежать путаницы и сделает вашу программу более ясной и понятной. Несомненно, вы

столкнетесь с программами, в которых квадратные скобки отсутствуют, так как некоторые все же считают, что в таком виде программа воспринимается лучше. Это, в общем, дело вкуса, но если вы выберете стиль адресации по одной ячейке памяти и будете содержательно его использовать, вам будет легче писать программы.

Вы можете использовать также такую форму адресации к памяти:

```
mov al,CharString[bx]
```

или даже

```
mov al,CharString[bx][si]+1
```

Все эти формы представляют собой то же самое, что размещение отдельных элементов адресации к памяти в одной паре квадратных скобок и разделение их знаком плюс. Таким образом, последний оператор эквивалентен оператору:

```
mov al,[charString+bx+si+1]
```

Здесь снова нужно выбрать ту форму записи, которая вам больше нравится, и придерживаться ее.

Квадратные скобки, в которые заключаются регистры, указывающие на ячейки памяти, являются обязательными. Без этих скобок, BX, например, интерпретируется, как операнд, а не как ссылка на операнд.

3.4 Ввод-вывод

Использовать базовые DOS для ввода и вывода строк крайне неудобно т.к. они обеспечивают только посимвольную обработку данных. И для ввода с клавиатуры одной строки нам пришлось бы организовывать цикл. Существует более эффективные способы обработки строк.

Рассмотрим пример использования команд условных переходов для обработки символов. Пусть мы вводим с клавиатуры некоторую строку символов (например,

имя файла), и хотим, чтобы в программе эта строка была записана прописными буквами, независимо от того, какие буквы использовались при ее вводе. Между прочим, при вводе с клавиатуры команд DOS система всегда выполняет эту операцию, поэтому и команды, и ключи, и имена файлов можно вводить как прописными, так и строчными буквами - DOS во всех случаях преобразует все буквы в прописные.

```
code segment
assume cs:code, ds:data
main proc
        mov ax,data             ;Инициализируем
        mov ds,ax               ;регистр DS
;Выведем служебное сообщение
        mov ah,09h              ;Функция вывода
        mov dx,offset msg ;адрес сообщения
        int 21h
;Поставим запрос к DOS на ввод строки
        mov ah,3Fh              ;функция ввода
        mov bx,0                ;дескриптор клавиатуры
        mov cx,80               ;ввод максимум 80 байт
        mov dx,offset buf  ;адрес буфера ввода
        int 21h
        mov actlen,ax           ;фактически введено
;Превратим строчные русские буквы в прописные
        mov cx,actlen           ;длина введенной строки
        mov si,0                ;указатель в буфере
filter: mov al,buf[si]      ;возьмем символ
        cmp al,'а'              ;меньше «а»?
        jb noletter             ;да, не преобразовывать
        cmp al,'я'              ;Больше «я»?
        ja noletter             ;да, не преобразовывать
        cmp al,'п'              ;больше «п»?
        ja more                 ;да, на дальнейшую программу
        sub al,20h              ;«а»..«п». преобразуем в прописную букву
        jmp store               ;на сохранение в буфер
more: cmp al,'р'                ;меньше «р»(псевдографика)?
        jb noletter             ;>п,<р не изменять
        sub al,50h              ; «р»..«я». Преобразуем в прописную букву
store: mov buf[si],al       ;отправим назад в buf
noletter:       inc si          ;сместим указатель
loop filter                     ;цикл по всем символам
```

```
;Выведем результат преобразования на экран
        mov ah,40h              ;функция ввода
        mov bx,1                ;дескриптор экрана
        moc cx,actlen           ;длина сообщения
        mov dx,offset buf  ;адрес сообщения
        int 21h
        mov ah,01               ;остановим программу
        int 21h                 ;в ожидании нажатия клавиши
 ;Завершим программу
        mov ax4C00h
        int 21h
main endp
code ends
data segment
msg db 'Вводите! $'
buf db 80 dup(' ')
actlen dw 0
data ends
stk segment stack
dw 128 dup (?)
stk ends
end main
```

В начале программы на экран выводится служебное сообщение "Вводите!", которое служит запросом программы, адресованным пользователю. Далее с помощью функции DOS 3Fh выполняется ввод строки текста с клавиатуры. Функция 3Fh может вводить данные из разных устройств - файлов, последовательного порта, клавиатуры.

Различные устройства идентифицируются их дескрипторами. При работе с файлами дескриптор каждого файла создается системой в процессе операции открытия или создания этого файла, а для стандартных устройств - клавиатуры, экрана, принтера и последовательного порта действуют дескрипторы, закрепляемые за этими устройствами при загрузке системы. Для ввода с клавиатуры используется дескриптор 0, для вывода на экран дескриптор.

При вызове функции 3Fh в регистр BX следует занести требуемый дескриптор, в регистр DX - адрес области в программе, выделенной для приема вводимых с клавиатуры символов, а в регистр CX - максимальное число вводимых символов.

Мы считаем, что пользователь не будет вводить более 80 символов. Можно ввести и меньше; в любом случае ввод строки следует завершить нажатием клавиши <Enter>. Функция 3Fh, отработав, вернет в регистре AX реальное число введенных символов (включая коды 13 и 10, образуемые при нажатии клавиши <Enter>). В примере 3.5 число введенных символов сохраняется в ячейке actlen с целью использования далее по ходу программы.

Далее в цикле из actlen шагов выполняется анализ каждого введенного символа путем сравнения с границами диапазонов строчных русских букв. Русские строчные буквы размещаются в двух диапазонах кодов ASCII (а...п и р...с), причем для преобразования в прописные букв первого диапазона их код следует уменьшать на 20h, а для преобразования букв второго диапазона - на 50h. Поэтому анализ проводится с помощью четырех команд сравнения cmp и соответствующих команд условных переходов. Модифицированный символ записывается на то же место в буфере buf.

После завершения анализа и преобразования введенных символов, выполняется контрольный вывод содержимого buf на экран. Поскольку мы заранее не знаем, сколько символов будет введено, вывод на экран осуществляется функцией 40h, среди параметров которой указывается число выводимых символов. Так же, как и в случае функции ввода 3Fh, для функции вывода 40h в регистре BX необходимо указать дескриптор устройства ввода, в данном случае экрана, а в регистре DX - адрес выводимой строки.

3.6 Задания

Во всех вариантах необходимо реализовать программу работы со строками. Исходная строка вводится с клавиатуры, результат выводится на экран. Слова в строке могут быть разделены пробелами и знаками препинания.

1) Найти слова, начинающиеся на заданную с клавиатуры букву, и перевернуть.
2) Найти слова, оканчивающиеся на заданную трехбуквенную комбинацию.
3) Отсортировать строку по длине слов.

4) Переставить слова – 6 слов: 1-6, 2-5, 3-4; 7 слов: 1-7, 2-6, 3-5, 4 на месте (цифры словами не считаются).
5) Подсчитать в строке количество слов, содержащих в середине гласную букву.
6) Отсортировать слова строки по третьей букве.
7) Найти слова, в которых больше трех повторяющихся символов.
8) Подсчитать количество слов-перевертышей в строке (шалаш).
9) Отсортировать слова в строки по алфавиту (по 1-ой букве, потом по 2-ой и т.д.).
10) Удалить в каждом слове строки повторяющиеся в нем буквы.
11) Вставить в каждое слово строки после заданного символа символ, введенный пользователем.
12) Найти слова, оканчивающиеся на заданную с клавиатуры букву, и перевернуть.
13) В каждом слове строки удвоить буквы.
14) Отсортировать буквы в каждом слове по алфавиту, оставляя позицию данного слова в строке неизменной.
15) Заменить все слова-перевертыши строки словом, введенным пользователем.
16) Удалить из каждого слова строки все гласные и вывести те гласные, которых не было в строке.
17) В каждом слове строки заменить 1-ую букву на последнюю, 2-ую на предпоследнюю и т.д.
18) Отсортировать слова в строке по количеству разных согласных букв, встречающихся в них.

4 РАБОТА С МАССИВАМИ И СТЕКОМ НА ЯЗЫКЕ АССЕМБЛЕРА

4.1 Общие сведения о массивах

Как структура представления, массив является упорядоченным множеством элементов определенного типа. Упорядоченность массива определяется набором целых чисел, называемых индексами, которые связываются с каждым элементом массива и однозначно конкретизируют его расположение среди других элементом массива. Локализация конкретного элемента массива - ключевая задача при разработке любых алгоритмов, работающих с массивами.

Наиболее просто представляются одномерные массивы. Соответствующая им структура хранения — это вектор. Она однозначна и есть не что иное, как просто последовательное расположение элементов в памяти. Чтобы локализовать нужный элемент одномерного массива, достаточно знать его индекс. Так как ассемблер не имеет средств для работы с массивом как структурой данных, то для доступа к элементу массива необходимо вычислить его адрес.

Представление двумерных массивов немного сложнее. Здесь мы имеем случай, когда структуры хранения и представления различны. О структуре представления говорить излишне — это матрица. Структура хранения остается прежней — вектор. Но теперь его нельзя без специальных оговорок интерпретировать однозначно. Все зависит от того, как решил разработчик программы «вытянуть» массив — по строкам или по столбцам. Наиболее естествен порядок расположения элементов массива — по строкам. При этом наиболее быстро изменяется последний элемент индекса.

4.2 Ввод – вывод массива

Пример ввода массива:

```
Read Proc
mov dl,10
mov ah,02h
int 21h
```

```
mov dl,13
mov ah,02h
int 21h ; перевод строки
mov ah,0ah
lea dx,len
int 21h         ; считали данные в len
mov ch,0
mov cl,len+1
mov si,offset val
mov bl,10
mov ax,0
L1:
mul bl
mov dl,[si]
sub dl,30h
mov dh,0
add ax,dx
inc si
LOOP L1 ; запускаем цикл
ret
Read EndP
```

Пример вывода массива:

```
Output Proc          ; вывод массива
mov bx,0
mov di,-1
n2:
inc pp
mov dh,pp
mov ah,02
mov bh,00
mov dl,00
int 10h
n1: inc di
mov dx,offset sr
mov ah,09h
int 21h
mov al,mas[bx][di]
PUSH di
xor ah,ah
mov kx,ax
viv:
```

```
 cmp al,0
 jge nosing
 neg al
 mov kx,ax
 mov dl,'-'
 mov ax,0200h
 int 21h
nosing:
 xor si,si
 xor dx,dx
 mov di,10
 mov ax,kx
iter1:
 div di
 xor dl,'0'
 mov str2[si],dl
 xor dx,dx
 inc si
 cmp ax,0
 ja iter1
 mov cx,si
 dec si
 mov ax,0200h
iter2:
 MOV dl,str2[si]
 int 21h
 dec si
 loop iter2
POP di
cmp di,jx
jb n1
mov di,-1
;Bi
Call OutBi
cmp bx,ix
jb n2
ret
Output EndP
```

4.3 Способы сортировки массивов.

Метод пузырька

```
Процедура bubble_sort
; сортирует массив слов методом пузырьковой сортировки
```

```
; ввод: DS:DI = адрес массива
; DX = размер массива (в словах)
bubble_sort proc near
 pusha
 cld
 cmp dx,1
 jbe sort_exit ; выйти, если сортировать нечего
 dec dx
sb_loop1:
 mov cx,dx ; установить длину цикла
 xor bx,bx ; BX будет флагом обмена
 mov si,di ; SI будет указателем на
 ; текущий элемент
sn_loop2:
 lodsw ; прочитать следующее слово
 cmp ax,word ptr [si]
 jbe no_swap ; если элементы не
 ; в порядке,
 xchg ax,word ptr [si] ; поменять их местами
 mov word ptr [si-2],ax
 inc bx ; и установить флаг в 1,
no_swap:
 loop on_loop2
 cmp bx,0 ; если сортировка не закончилась,
 jne sn_loop1 ; перейти к следующему элементу
sort_exit:
 popa
 ret
bubble_sort endp
```

Пузырьковая сортировка осуществляется так медленно потому, что сравнения выполняются лишь между соседними элементами. Чтобы получить более быстрый метод сортировки перестановкой, следует выполнять сравнение и перестановку элементов, отстоящих далеко друг от друга. На этой идее основан алгоритм, который называется «быстрая сортировка». Он работает следующим образом: делается предположение, что первый элемент является средним по отношению к остальным. На основе такого предположения все элементы разбиваются на две группы - больше и меньше предполагаемого среднего. Затем обе группы отдельно сортируются таким же методом. В худшем случае быстрая сортировка массива из N элементов требует

N^2 операций, но в среднем случае - только $2n*\log_2 n$ сравнений и еще меньшее число перестановок.

Метод быстрой сортировки:

```
; Процедура quick_sort
; сортирует массив слов методом быстрой сортировки
; ввод: DS:BX = адрес массива
; DX = число элементов массива
quicksort proc near
cmp dx,1 ; Если число элементов 1 или 0,
jle qsort_done ; то сортировка уже закончилась
xor di,di ; индекс для просмотра сверху (DI = 0)
mov si,dx ; индекс для просмотра снизу (SI = DX)
dec si ; SI = DX-1,          так как элементы нумеруются с нуля,
shl si,1 ; и умножить на 2,                 так как это массив слов
mov ax,word ptr [bx] ; AX = элемент X1, объявленный средним
step_2: ; просмотр массива снизу, пока не встретится
 ; элемент, меньший или равный X1
cmp word ptr [bx][si],ax ; сравнить XDI и X1
jle step_3      ; если XSI больше,
sub si,2        ; перейти к следующему снизу элементу
jmp short step_2     ; и продолжить просмотр
step_3:                         ; просмотр массива сверху, пока не встретится
 ; элемент меньше X1              или оба просмотра не придут
 ; в одну точку
cmp si,di ; если просмотры встретились,
je step_5               ; перейти к шагу 5,
add di,2 ; иначе: перейти
                        ; к следующему сверху элементу,
cmp word ptr [bx][di],ax          ; если он меньше X1,
jl step_3               ; продолжить шаг 3
step_4:
; DI указывает на элемент, который не должен быть
; в верхней части, SI указывает на элемент,
; который не должен быть в нижней. Поменять их местами
mov cx,word ptr [bx][di] ; CX = XDI
xchg cx,word ptr [bx][si] ; CX = XSI, XSI = XDI
mov word ptr [bx][di],cx ; XDI = CX
jmp short step_2
step_5: ; Просмотры встретились. Все элементы в нижней
 ; группе больше X1, все элементы в верхней группе
 ; и текущий - меньше или равны X1 Осталось
 ; поменять местами X1 и текущий элемент:
xchg ax,word ptr [bx][di] ; AX = XDI, XDI = X1
```

```
mov word ptr [bx],ax ; X1 = AX
; теперь можно отсортировать каждую из полученных групп
push dx
push di
push bx

mov dx,di ; длина массива X1...XDI-1
 shr dx,1 ; в DX
 call quick_sort ; сортировка

 pop bx
 pop di
 pop dx
 add bx,di ; начало массива XDI+1...XN
 add bx,2 ; в BX
 shr di,1 ; длина массива XDI+1...XN
 inc di
 sub dx,di ; в DX
 call quicksort ; сортировка
qsort_done: ret
quicksort endp
```

Кроме того, что быстрая сортировка - самый известный пример алгоритма, использующего рекурсию, то есть вызывающего самого себя. Это еще и самая быстрая из сортировок «на месте», то есть сортировка, использующая только ту память, в которой хранятся элементы сортируемого массива. Можно доказать, что сортировку нельзя выполнить быстрее, чем за $n*\log_2 n$ операций, ни в худшем, ни в среднем случаях; и быстрая сортировка достаточно хорошо приближается к этому пределу в среднем случае. Сортировки, достигающие теоретического предела, тоже существуют — это сортировки турнирным выбором и сортировки вставлением в сбалансированные деревья, но для их работы требуется резервирование дополнительной памяти Так что, например, работа со сбалансированными деревьями будет происходить медленно из-за дополнительных затрат на поддержку сложных структур данных в памяти.

4.4 Работа со стеком в ассемблере

4.4.1 Команды работы со стеком

В процессорах Intel команду BSWAP можно использовать и для обращения порядка байт в 16-битных регистрах, но в некоторых совместимых процессорах других фирм этот вариант BSWAP не реализован.

Команда:	**PUSH** источник
Назначение:	Поместить данные в стек
Процессор:	8086

Команда помещает содержимое источника в стек. В качестве параметра «источник» может быть регистр, сегментный регистр, непосредственный операнд или переменная. Фактически эта команда копирует содержимое источника в память по адресу SS:[ESP] и уменьшает ESP на размер источника в байтах (2 или 4). Команда PUSH практически всегда используется в паре с POP (считать данные из стека). Так, например, чтобы скопировать содержимое одного сегментного регистра в другой (что нельзя выполнить одной командой MOV), можно использовать такую последовательность команд:

```
push cs
pop ds ; теперь DS указывает на тот же сегмент, что и CS
```

Другое частое применение команд PUSH/POP — временное хранение переменных, например:

```
push ax ; сохраняет текущее значение AX
... ; здесь располагаются какие-нибудь команды,
; которые используют AX, например CMPXCHG
pop ax ; восстанавливает старое значение AX
```

Начиная с 80286, команда PUSH ESP (или SP) помещает в стек значение ESP до того, как эта же команда его уменьшит, в то время как на 8086 SP помещался в стек уже уменьшенным на два.

Команда:	**POP** приемник
Назначение:	Считать данные из стека
Процессор:	8086

Команда помещает в приемник слово или двойное слово, находящееся в вершине стека, увеличивая ESP на 2 или 4 соответственно. POP выполняет действие, полностью обратное PUSH. Приемником может быть регистр общего назначения, сегментный регистр, кроме CS (чтобы загрузить CS из стека, надо воспользоваться командой RET), или переменная. Если в роли приемника выступает операнд, использующий ESP для косвенной адресации, команда POP вычисляет адрес операнда уже после того, как она увеличивает ESP.

Команда:	**PUSHA** **PUSHAD**
Назначение:	Поместить в стек все регистры общего назначения
Процессор:	80186 80386

PUSHA помещает в стек регистры в следующем порядке: AX, CX, DX, BX, SP, BP, SI и DI. PUSHAD помещает в стек EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI. (В случае SP и ESP используется значение, которое находилось в этом регистре до начала работы команды.) В паре с командами POPA/POPAD, считывающими эти же регистры из стека в обратном порядке, это позволяет писать подпрограммы (обычно обработчики прерываний), которые не должны изменять значения регистров по окончании своей работы. В начале такой подпрограммы вызывают команду PUSHA, а в конце — POPA.

На самом деле PUSHA и PUSHAD — одна и та же команда с кодом 60h. Ее поведение определяется тем, выполняется ли она в 16- или в 32-битном режиме. Если программист использует команду PUSHAD в 16-битном сегменте или PUSHA в 32-битном, ассемблер просто записывает перед ней префикс изменения размерности операнда (66h).

Это же будет распространяться на некоторые другие пары команд: POPA/POPAD, POPF/POPFD, PUSHF/PUSHFD, JCXZ/JECXZ, CMPSW/CMPSD, INSW/INSD, LODSW/LODSD, MOVSW/MOVSD, OUTSW/OUTSD, SCASW/SCASD и STOSW/STOSD.

Команда:	**POPA** **POPAD**
Назначение:	Загрузить из стека все регистры общего назначения
Процессор:	80186 80386

4.4.2 Передача параметров в стеке

Параметры помещаются в стек сразу перед вызовом процедуры. Именно этот метод используют языки высокого уровня, такие как C и Pascal. Для чтения параметров из стека обычно используют не команду POP, а регистр BP, в который помещают адрес вершины стека после входа в процедуру:

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

Параметры в стеке, адрес возврата и старое значение BP вместе называются активационной записью функции.

Для удобства ссылок на параметры, переданные в стеке, внутри функции иногда используют директивы EQU, чтобы не писать каждый раз точное смещение параметра от начала активационной записи (то есть от BP), например так:

```
 push X
 push Y
 push Z
 call xyzzy
 [...]
xyzzy proc near
xyzzy_z equ [bp+8]
xyzzy_y equ [bp+6]
xyzzy_x equ [bp+4]
 push bp
 mov bp,sp
(команды, которые могут использовать стек)
 mov ax,xyzzy_x ;считать параметр X
(остальные команды)
 pop bp
 ret 6
xyzzy endp
```

При внимательном анализе этого метода передачи параметров возникает сразу два вопроса: кто должен удалять параметры из стека, процедура или вызывающая ее программа, и в каком порядке помещать параметры в стек. В обоих случаях оказывается, что оба варианта имеют свои «за» и «против», так, например, если стек освобождает процедура (командой RET число_байтов), то код программы получается меньшим, а если за освобождение стека от параметров отвечает вызывающая функция, как в нашем примере, то становится возможным вызвать несколько функций с одними и теми же параметрами просто последовательными командами CALL. Первый способ, более строгий, используется при реализации процедур в языке Pascal, а второй, дающий больше возможностей для оптимизации, - в языке C. Разумеется, если передача параметров через стек применяется и для возврата результатов работы процедуры, из стека не надо удалять все параметры, но популярные языки высокого уровня не пользуются этим методом. Кроме того, в языке C параметры помещают в стек в обратном порядке (справа налево), так что

становятся возможными функции с изменяемым числом параметров (как, например, printf - первый параметр, считываемый из [BP+4], определяет число остальных параметров). Но подробнее о тонкостях передачи параметров в стеке рассказано далее, а здесь приведен обзор методов.

4.4.3 Передача параметров в потоке кода

В этом необычном методе передаваемые процедуре данные размещаются прямо в коде программы, сразу после команды CALL (как реализована процедура print в одной из стандартных библиотек процедур для ассемблера UCRLIB):

```
call print
db "This ASCIZ-line will be printed",0
(следующая команда)
```

Чтобы прочитать параметр, процедура должна использовать его адрес, который автоматически передается в стеке как адрес возврата из процедуры. Разумеется, функция должна будет изменить адрес возврата на первый байт после конца переданных параметров перед выполнением команды RET. Например, процедуру print можно реализовать следующим образом:

```
print proc near
 push bp
 mov bp,sp
 push ax
 push si
 mov si,[bp+2]              ; прочитать адрес
                ; возврата/начала данных
 cld            ; установить флаг направления
                ; для команды lodsb
print_readchar:
 lodsb                  ; прочитать байт из строки,
 test al,al             ; если это 0 (конец строки),
 jz print_done                  ; вывод строки закончен
 int 29h                ; вывести символ в AL на экран
 jmp short print_readchar
print_done:
 mov [bp+2],si              ; поместить новый адрес возврата в стек
 pop si
 pop ax
 pop bp
 ret
```

```
print endp
```

Передача параметров в потоке кода, так же, как и передача параметров в стеке в обратном порядке (справа налево), позволяет передавать различное число параметров. Но этот метод - единственный, позволяющий передать по значению параметр различной длины, что и продемонстрировал этот пример. Доступ к параметрам, переданным в потоке кода, несколько медленнее, чем к параметрам, переданным в регистрах, глобальных переменных или стеке, и примерно совпадает со следующим методом.

4.5 Задания

Дан двумерный массив. Размер массива и его элементы вводятся пользователем с клавиатуры. Результат работы программы выводится на экран.

Варианты:

1) Посчитать и вывести сумму положительных элементов, расположенных под главной диагональю матрицы.
2) Заменить элементы главной диагонали на суммы элементов соответствующих строк, затем отсортировать получившийся массив по возрастанию элементов главной диагонали.
3) Решить систему линейных уравнений методом Гаусса, в случае несуществования решения вывести сообщение.
4) Посчитать и вывести суммы элементов больше заданного пользователем числа, расположенных в верхнем и нижнем треугольниках, образуемых диагоналями. Заменить элементы кратные 3 левого треугольника на первую сумму, а кратные 5 на вторую.
5) Отсортировать массив по возрастанию элементов побочной диагонали, затем поменять местами элементы, расположенные над и под побочной диагональю.
6) Отсортировать строки матрицы по суммам модулей элементов и вывести столбец с максимальным количеством отрицательных элементов.
7) Подсчитать и вывести максимальный диагональный минор матрицы.

8) Отсортировать элементы матрицы по возрастанию среднего арифметического по столбцу.
9) Найти произведение матриц произвольного размера (должна производиться проверка возможности выполнения операции умножения).
10) Посчитать и вывести обратную матрицу методом Гаусса.
11) Найти в верхнем треугольнике, образуемом диагоналями максимальный элемент, а в нижнем минимальный. Затем увеличить минимальный элемент на 30%, а максимальный уменьшить на 30% от разности этих элементов. Повторять процедуру до тех пор, пока разность между элементами превышает 10%.
12) Найти и вывести определитель матрицы. Все элементы матрицы, значение которых больше определителя уменьшить на 20%, элементы, значение которых меньше увеличить на 30%.
13) В матрице поменять местами элементы треугольников, образованных диагоналями, заданное пользователем количество раз по часовой стрелке

    Пример:

    Исходная матрица

    1 2 3 4 5

    6 7 8 9 10

    11 12 13 14 15

    16 17 18 19 20

    21 22 23 24 25

    Результат (количество раз - 2)

    1 24 23 22 5

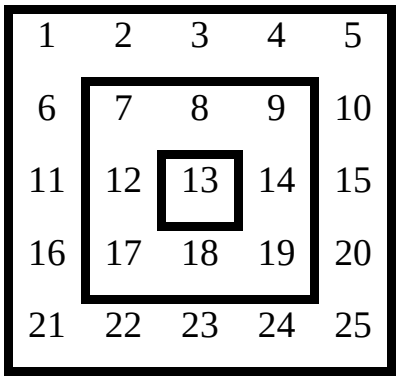
    20 7 18 9 6

    15 14 13 12 11

    10 17 8 19 16

    21 4 3 2 25

14) Повернуть элементы матрицы внутренний ряд по часовой стрелке, следующий против и т.д.



15) Найти максимум по каждому столбцу, затем вычесть из него n-1 (n количество элементов в массиве), к каждому элементу столбца прибавить 1. Найти минимум по строке, затем прибавить к нему n-1 (n количество элементов в массиве), из каждого элемента строки вычесть 1.
16) Сортировать каждый столбец по возрастанию, а строку по убыванию до тех пор, пока на очередном шаге итерации матрица останется неизменной, но не более 5 итераций.

5 РАБОТА С МАТЕМАТИЧЕСКИМ СОПРОЦЕССОРОМ В СРЕДЕ ASSEMBLER

5.1 Основные сведения

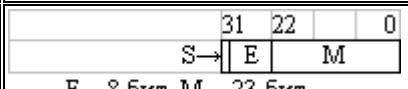
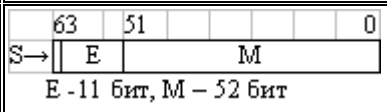
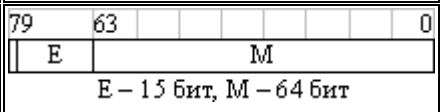
Сопроцессор - это специализированная интегральная схема, которая работает в содружестве с ЦП, но менее универсальна. Сопроцессор предназначен для выполнения специфического набора функций, например: выполнение операций с вещественными числами - математический сопроцессор, подготовка графических изображений и трехмерных сцен - графический сопроцессор, цифровая обработка сигналов - сигнальный сопроцессор и др.

Один из наиболее распространенных типов сопроцессоров - математический сопроцессор. Математический сопроцессор предназначен для быстрого выполнения арифметических операций с плавающей точкой, предоставления часто используемых вещественных констант (π, log210, log2e, ln2, :), вычисления тригонометрических и прочих трансцендентных функций (tg, arctg, log, ...).

Большинство современных математических сопроцессоров для представления вещественных чисел используют стандарт IEEE 754-1985 "IEEE1) Standard for Binary Floating-Point Arithmetics". Старший разряд двоичного представления вещественного числа всегда кодирует знак числа. Остальная часть разбивается на две части: экспоненту и мантиссу. Вещественное число вычисляется как: $(-1)S \cdot 2E \cdot M$, где S - знаковый бит числа, E - экспонента, M - мантисса. Если $1 \leq M < 2$, то такое число называется нормализованным. При хранении нормализованных чисел сопроцессор отбрасывает целую часть мантиссы (она всегда 1), сохраняя лишь дробную часть. Экспонента кодируется со сдвигом на половину разрядной сетки, таким образом, удается избежать вопроса о кодировании знака экспоненты. Т.е. при 8-битной разрядности экспоненты код 0 соответствует числу -127, 1 - числу -126, ..., 255 числу +126 (экспонента вычисляется как код 127).

Стандарт IEEE-754 определяет три основных способа кодирования (типа) вещественных чисел.

Таблица 2 - Типы (способы кодирования) вещественных чисел

Тип	**Диапазон значений (по модулю)**	**Двоичное представление**
вещественное ординарной точности (single precision) - 32 бит	$1{,}18\cdot10^{-38}$... $3{,}40\cdot10^{38}$	
вещественное двойной точности (double precision) - 64 бит	$2{,}23\cdot10^{-308}$... $1{,}79\cdot10^{308}$	
вещественное расширенной точности (extended precision) - 80 бит	$3{,}37\cdot10^{-4932}$... $1{,}18\cdot10^{4392}$	

Если результат численной операции не может быть точно представлен в выбранном формате, сопроцессор выполняет округление в соответствии с полем RC (таблица 3). По умолчанию RC = 00.

Таблица 3 - Режимы округления сопроцессоров Intel x87

RC	Режим	Пример 1 $1{,}000E_21 < 2{,}23E_{10}0 < 1{,}001E_21$ $-1{,}001E_21 < -2{,}23E_{10}0 < -1{,}000E_21$	Пример 2 $1{,}000E_21 < 2{,}05E_{10}0 < 1{,}001E_21$ $-1{,}001E_21 < -2{,}05E_{10}0 < -1{,}000E_21$
00	Округление к ближайшему (или четному)	$2{,}23E_{10}0 \approx 1{,}001E_21$ $-2{,}23E_{10}0 \approx -1{,}001E_21$	$2{,}05E_{10}0 \approx 1{,}000E_21$ $-2{,}05E_{10}0 \approx -1{,}000E_21$
01	Округление вниз (к ∞)	$2{,}23E_{10}0 \approx 1{,}000E_21$ $-2{,}23E_{10}0 \approx -1{,}001E_21$	$2{,}05E_{10}0 \approx 1{,}000E_21$ $-2{,}05E_{10}0 \approx -1{,}001E_21$
10	Округление вверх (к +∞)	$2{,}23E_{10}0 \approx 1{,}001E_21$ $-2{,}23E_{10}0 \approx -1{,}000E_21$	$2{,}05E_{10}0 \approx 1{,}001E_21$ $-2{,}05E_{10}0 \approx -1{,}000E_21$
11	Округление к нулю (усечение)	$2{,}23E_{10}0 \approx 1{,}000E_21$ $-2{,}23E_{10}0 \approx -1{,}000E_21$	$2{,}05E_{10}0 \approx 1{,}000E_21$ $-2{,}05E_{10}0 \approx -1{,}000E_21$

5.2 Команды сопроцессора

5.2.1 Команды пересылки данных

Команды загрузки в стек (Fpu LoaD):

- FLD - загружает из памяти в вершину стека ST(0) вещественное число
- FILD - загружает из памяти в вершину стека ST(0) целое число
- FBLD - загружает из памяти в вершину стека ST(0) двоично-десятичное число

Команды сохранения и извлечения из стека (Fpu STore and Pop):

- FSTP память - извлекает из вершины стека ST(0) в память вещественное число
- FISTP память - извлекает из вершины стека ST(0) в память целое число
- FBSTP память - извлекает из вершины стека ST(0) в память двоично-десятичное число

Эти команды сначала сохраняют вершину стека в памяти, а потом удаляют данные из вершины стека.

Команды копирования данных (Fpu STore):

- FST память - извлекает из вершины стека ST(0) в память вещественное число
- FIST память - извлекает из вершины стека ST(0) в память целое число
- FBST память - извлекает из вершины стека ST(0) в память двоично-десятичное число

Команда обмена (Fpu eXCHange):

- FXCH - обмен содержимым верхушки стека ST(0) и численного регистра, указанного в качестве операнда команды

5.2.2 Арифметические команды

Сопроцессор использует шесть основных типов арифметических команд:

- Fxxx

Первый операнд берется из верхушки стека (источник), второй - следующий элемент стека. Результат выполнения команды записывается в стек

- Fxxx память

Источник берется из памяти, приемником является верхушка стека ST(0). Указатель стека ST не изменяется, команда действительна только для операндов с одинарной и двойной точностью

- Fixxx память

Аналогично предыдущему типу команды, но операндами могут быть 16- или 32-разрядные целые числа

- Fxxx ST, ST(i)

Для этого типа регистр ST(i) является источником, а ST(0) - верхушка стека - приемником. Указатель стека не изменяется

- Fxxx ST(i), ST

Для этого типа регистр ST(0) является источником, а ST(i) - приемником. Указатель стека не изменяется

- FxxxP ST(i), ST

Регистр ST(i) - приемник, регистр ST(0) - источник. После выполнения команды источник ST(0) извлекается из стека

Строка "xxx" может принимать следующие значения:

- ADD - Сложение
- SUB - Вычитание
- SUBR - Обратное вычитание, уменьшаемое и вычитаемое меняются местами
- MUL - Умножение
- DIV - Деление
- DIVR - Обратное деление, делимое и делитель меняются местами

Кроме основных арифметических команд имеются дополнительные арифметические команды:

- FSQRT - Извлечение квадратного корня
- FSCALE - Масштабирование на степень числа 2

- FPREM - Вычисление частичного остатка
- FRNDINT - Округление до целого
- FXTRACT - Выделение порядка числа и мантиссы
- FABS - Вычисление абсолютной величины числа
- FCHS - Изменение знака числа

По команде FSQRT вычисленное значение квадратного корня записывается в верхушку стека ST(0).

Команда FSCALE изменяет порядок числа, находящегося в ST(0). По этой команде значение порядка числа ST(0) складывается с масштабным коэффициентом, который должен быть предварительно записан в ST(1). Действие этой команды можно представить следующей формулой:

ST(0) = ST(0) * 2n, где -215 <= n <= +215

В этой формуле n - это ST(1).

Команда FPREM вычисляет остаток от деления делимого ST(0) на делитель ST(1). Знак результата равен знаку ST(0), а сам результат получается в вершине стека ST(0).

Действие команды заключается в сдвигах и вычитаниях, аналогично ручному делению "в столбик". После выполнения команды флаг C2 регистра состояния может принимать следующие значения:

- 0 - Остаток от деления, полученный в ST(0), меньше делителя ST(1), команда завершилась полностью
- 1 - ST(0) содержит частичный остаток, программа должна еще раз выполнить команду для получения точного значения остатка

Команда RNDINT округляет ST(0) в соответствии с содержимым поля RC управляющего регистра.

Команда FABS вычисляет абсолютное значение ST(0). Аналогично, команда FCHS изменяет знак ST(0) на противоположный.

Трансцендентные команды

Трансцендентные команды предназначены для вычисления следующих функций:

- тригонометрические (sin, cos, tg,...)
- обратные тригонометрические (arcsin, arccos,...)
- показательные (xy, 2x, 10x, ex)
- гиперболические (sh, ch, th,...)
- обратные гиперболические (arsh, arch, arcth,...)

Вот список всех трансцендентных команд математического сопроцессора:

- FPTAN Вычисление частичного тангенса
- FPATAN Вычисление частичного арктангенса
- FYL2X Вычисление y*log2(x)
- FYL2XP1 Вычисление y*log2(x+1)
- F2XM1 Вычисление 2x-1
- FCOS Вычисление cos(x)
- FSIN Вычисление sin(x)
- FSINCOS Вычисление sin(x) и cos(x) одновременно

Команда FPTAN вычисляет частичный тангенс ST(0), размещая в стеке такие два числа x и y, что y/x = tg(ST(0)).

После выполнения команды число y располагается в ST(0), а число x включается в стек сверху (то есть записывается в ST(1)). Аргумент команды FPTAN должен находится в пределах:

0 <= ST(0) <= pi/4

Пользуясь полученным значением частичного тангенса, можно вычислить другие тригонометрические функции по следующим формулам:

- sin(z) = 2*(y/x) / (1 + (y/x)2)
- cos(z) = (1 - (y/x)2) / (1 + (y/x)2)
- tg(z/2) = y/x;
- ctg(z/2) = x/y;
- cosec(z) = (1 + (y/x)2) / 2*(y/x)
- sec(z) = (1 + (y/x)2) / (1 - (y/x)2)

Где z - значение, находившееся в ST(0) до выполнения команды FPTAN, x и y - значения в регистрах ST(0) и ST(1), соответственно.

Команда FPATAN вычисляет частичный арктангенс:

z=arctg(ST(0)/ST(1))=arctg(x/y).

Перед выполнением команды числа x и y располагаются в ST(0) и ST(1), соответственно. Аргументы команды FPATAN должен находится в пределах:

0 < y < x

Результат записывается в ST(0).

Команда FYL2X вычисляет выражение y*log2(x), операнды x и y размещаются, соответственно, в ST(0) и ST(1). Операнды извлекаются из стека, а результат записывается в стек. параметр x должен быть положительным числом.

Пользуясь результатом выполнения этой команды, можно вычислить следующим образом логарифмические функции:

- Логарифм по основанию два: log2(x) = FYL2(x)
- Натуральный логарифм: loge(x) = loge(2) * log2(x) = FYL2X(loge(2), x) = FYL2X(FLDLN2, x)
- Десятичный логарифм: log10(x) = log10(2) * log2(x) = FYL2X (log10(2), x) = FYL2X(FLDLG2, x)

Функция FYL2XP1 вычисляет выражение y*log2(x+1), где x соответствует ST(0), а y - ST(1). Результат записывается в ST(0), оба операнда выталкиваются из стека и теряются.

На операнд x накладывается ограничение: 0 < x < 1 - 1/sqrt(2)

Команда F2XM1 вычисляет выражение 2x-1, где x - ST(0). Результат записывается в ST(0), параметр должен находиться в следующих пределах: 0 <= x <= 0,5

Команда FCOS вычисляет cos(x). Параметр x должен находиться в ST(0), туда же записывается результат выполнения команды.

Команда FSIN аналогична команде FCOS, но вычисляет значение синуса ST(0).

Команда FSINCOS вычисляет одновременно значения синуса и косинуса параметра ST(0). Значение синуса записывается в ST(1), косинуса - в ST(0).

Пример ввода вещественного числа в математический сопроцессор (числа не более 1000, количество знаков после запятой не больше 4)

5.3 Пример

```
.286
.model small
.data
 buf_st db 10,0
 buf db 10 dup (0)
 c dw 0 ;целая часть числа
 d dw 0 ;дробная часть числа
 p dw 1 ;порядок числа
 res dd ?

.stack 128
.code
 .startup
 mov ah, 0ah
 lea dx, buf_st
 int 21h
 lea si,[buf]
l_celoe:
 cmp byte ptr [si], 13
 je l_end
 cmp byte ptr [si], '.'
 je l_drob

 mov bl, [si]
 sub bl, 30h
 mov bh, 0
 mov cx, 10
 mov ax, c
 mul cx
 add ax, bx
 mov c, ax
 inc si
 jmp l_celoe
l_drob:
 inc si
 cmp byte ptr [si], 13
 je l_end
 mov bl, [si]
```

```
 sub bl, 30h
 mov bh, 0
 mov cx, 10
 mov ax, d
 mul cx
 add ax, bx
 mov d, ax
 mov ax, p
 mul cx
 mov p, ax
 jmp l_drob
l_end:
 finit
 fild d
 fidiv p
 fiadd c
 fst res
 fwait

 mov ax, 4c00h
 int 21h
end
```

5.4 Задания

Рассчитать и вывести значение выражения, при заданных пользователем значениях x и a.

1) $$y = \frac{-15\ /\ 6 + (34\ -\ x * 2) * 2}{16\ +\ \sin(\ a\ )\ /\ 8} - 14$$

2) $$y = \frac{\frac{37\ +\ 14\ *\ 5}{\cos(\ 4\ )} + 5 * x * 6}{17\ -\ 4\ /\ 2} + 5 * 4 * e^{a}$$

3) $$y = 17\ *\ 4 + \frac{5 * a\ -\ 8\ /\ 3}{\frac{14\ *\ \sin(\ x\ )\ +\ 5}{4} + \frac{16\ *\ 6\ +\ 5}{10}}$$

4) $$y = 35\ *\ 4 + \frac{\frac{17\ -\ 8\ /(3\ +\ \cos(\ x\ ))}{2} + 4 * 8}{3\ +\ 2\ *\ e^{a}}$$

5) $$y = 3 * 6 + 4 * 4\ /(\sin(\ a\ ) * 3) * (48\ +\ 16\ /\ 2) * \frac{5\ +\ 8\ *\ \cos(\ x\ )}{14\ -\ 3}$$

6) $$y = \frac{14 - 4}{2} * (5 * \sin(x - a) + 16 / 4) * \frac{17 + 8 / 3}{\frac{8}{4} * (1 + e^{2}) + 2 * 5}$$

7) $$y = \frac{171 + 4 * 3 * 2}{\frac{36 * (\sin(x - e^{a})) - 4}{3} - 17} + (5 * x + 6) * 8 * \frac{17}{5 + 2 - a}$$

8) $$y = \frac{250 - 4 * 6}{\frac{33 * \cos(x + a) + 4}{2} - 17} * (2 + 4 - \sin(a) * e^{x}) - 17$$

9) $$y = 7 + 4 * 6 + 5 /(2 - \sin(x + \frac{a}{3})) * 13 + \frac{\frac{14 - 4}{2} * e^{x-a} + 17 * 6}{\frac{36 - 4}{3} + 2}$$

10) $$y = \frac{3 * 4 + 5 / 7}{5 - (3 - 4 * \sin(\frac{x}{5} + a)) / 2} - \frac{17 * \cos(a) + 2}{3 * 6 + 6 * \ln(2)}$$

11) $$y = \frac{25 / 4 + 5 * 6}{36 - e^{x*a}} + \frac{37 * \sin(a + x)}{\frac{8 * 6 + 5}{3} + 3 * 2}$$

12) $$y = \frac{37 / 2 + \frac{4 * 6 + 5 * e^{x-a}}{4}}{15 - 7} * (5 + 8 * \frac{\sin(x)}{\cos(a)}) - 136$$

13) $$y = \frac{3 * 4 + 5 * 6 - e^{x} * a}{7 / 2 + 4 / 3 + 12 / 3} * \frac{16}{2} 3 * 6 * \cos(\frac{a}{x})$$

14) $$y = \frac{49 / 7 + 42 / 6 + 6 * 2}{8 * 5 / 9 - 4 * 6} * (\cos(x) - \sin(a)) + \frac{23 * e^{x/a} + 12}{26 - 4} * 5$$

15) $$y = \frac{\frac{36 + 4 * 5}{2 + 6 * \cos(x - a)} - 14 / 7 + 1}{\frac{36 - 4}{2} + 12 * \sin(x + a)} - 7 * 6 * \frac{3 * e^{x*a} + 4}{12 - 7}$$

16) $$y = \frac{\frac{2 + 3}{2} * \cos(x) + \frac{4 * 6 + 25 * 5}{36 - 4}}{\frac{250 * 2 - 14}{37} * \sin(x - a)} + (12 - 4) * (37 - 8) * \ln(x)$$

17) $$y = \frac{\frac{320}{4} + \frac{256}{2} - 14 * \frac{\sin(x - a)}{\cos(x + a)}}{56 * 2 * \lg(x - a)} * (3 + 2)$$

18) $$y = \frac{(12 + 4 - 7) * 5}{36 * 4 - \frac{17}{2} * e^{x-a}} * \frac{(2 + 5)}{7} * \cos(x + a) + 14 * \ln(a)$$

19) $$y = \frac{34 + 260 / 2 + 170 * 3 * e^{x*a}}{\frac{36 + 8}{2} * \ln(x + a) + 5 * 6} - \frac{34}{2} * (6 * \cos(x * a) + 7)$$

6 ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА ЗАДАЧ С ИСПОЛЬЗОВАНИЕМ СИСТЕМНЫХ РЕСУРСОВ BIOS. РАБОТА В ГРАФИЧЕСКОМ РЕЖИМЕ

6.1 Графический режим

Для генерации цветных изображений в графическом режиме используются минимальные точки растра - пиксели или пэлы (pixel).

При среднем разрешении каждый байт представляет четыре точки, пронумерованных от 0 до 3:

Байт: |C1 C0|C1 C0|C1 C0|C1 C0|

Пиксели: 0 1 2 3

В любой момент для каждой точки возможны четыре цвета, от 0 до 3. Ограничение в 4 цвета объясняется тем, что двухбитовая точка имеет 4 комбинации значений битов: 00, 01, 10 и 11. Можно выбрать значение 00 для любого из 16 возможных цветов фона или выбрать значение 01, 10, и 11 для одной из двух палитр. Каждая палитра имеет три цвета:

C1 C0 Палитра 0 Палитра 1

0 0 фон фон

0 1 зеленый голубой

1 0 красный сиреневый

1 1 коричневый белый

Для выбора цвета палитры и фона используется INT 10H. Таким образом, если, например, выбран фон желтого цвета и палитра 0, то возможны следующие цвета точки: желтый, зеленый, красный и коричневый. Байт, содержащий значение 10101010, соответствует красным точкам. Если выбрать цвет фона - синий и палитру 1, то возможные цвета: синий, голубой, сиреневый и белый. Байт, содержащий значение 00011011, отображает синюю, голубую, сиреневую и белую точки.

6.2 Прерывание BIOS INT 10H для графики

Функция AH=00 команды INT 10H устанавливает графический режим. Функция AH=11 команды INT 10H позволяет выбрать цвет палитры и вывести на экран графический символ. Код в регистре AH определяет функцию:

AH=00: Установка режима. Нулевое значение в регистре AH и 04 в регистре AL устанавливают стандартный цветной графический режим:

```
MOV AH,00 ;Функция установки режима
MOV AL,04 ;Разрешение 320x200
INT 10H
```

Установка графического режима приводит к исчезновению курсора с экрана. Подробности по установке режима приведены в главе 9.

AH=0BH: Установка цветовой палитры. Число в регистре BH определяет назначение регистра BL:

BH=00 выбирает цвета фона и бордюра в соответствии с содержимым регистра BL. Цвет фона от 1 до 16 соответствует шест. значениям от 0 до F; BH=01 выбирает палитру соответственно содержимому регистра BL (0 или 1):

```
MOV AH,0BH ;Функция установки цвета
MOV BH,01 ;Выбор палитры
MOV BL,00 ; 0 (зеленый, красный, корич.)
INT 10H ;Вызвать BIOS
```

Палитра, установленная один раз, сохраняется, пока не будет отменена другой командой. При смене палитры весь экран меняет цветовую комбинацию. При использовании функции AH=0BH в текстовом режиме, значение, установленное для цвета 0 в палитре, определяет цвет бордюра.

AH=0CH: Вывод точки на экран. Использование кода 0C в регистре AH позволяет вывести на экран точку в выбранном цвете (фон и палитра). Например, для разрешения 320x200 загрузим в регистр DX вертикальную координату (от 0 до 199), а в регистр CX - горизонтальную координату (от 0 до 319). В регистр AL поместим цвет точки (от 0 до 3):

MOV AH,0CH ;Функция вывода точки

MOV AL,цвет ;Цвет точки

MOV CX,столбец ;Горизонтальная координата

MOV DX,строка ;Вертикальная координата

INT 10H ;Вызвать BIOS

AH=0DH: Чтение точки с экрана. Данная функция позволяет прочитать точку для определения ее цвета. В регистр DX должна быть загружена вертикальная координата (от 0 до 199), а в регистр CX - горизонтальная (от 0 до 319). В регистре AH должно быть значение 0D. Функция возвращает цвет точки в регистре AL.

6.3 Задания

Построить и визуализировать график функции согласно варианту. Для расчета значения функции использовать математический сопроцессор. Должны быть также визуализированы оси координат.

Варианты заданий:

1) $y_x := \frac{13\cdot\cos(x) + 12\cdot\sin(x)}{e^x} - \sqrt{x^3 - x^2 + 45}$

2) $y_x := \sqrt{x^2 + 12\cdot x - 45} - x\cdot e^x$

3) $y_x := (\cos(x) - \sin(x))^2 - (\cos(x) + \sin(x))^3$

4) $y_x := \left|\frac{1}{x^2 + 1} - \ln(x)\right|\cdot\left(e^x\cdot\cos(x)\right)$

5) $y_x := \frac{\cos(x)^2 - \sin(x)^2}{\sin(x) + \cos(x)}\cdot e^{\frac{1}{x}}$

6) $y_x := \left(15\cdot x^3 + 12\cdot x^2 + 17\cdot x + 34\right)\cdot\frac{1}{x^2 - 23}$

7) $y_x := \frac{\ln\left(x + \sqrt{x^2 - 18\cdot x}\right)}{\log\left(x + e^{x^2 - 12}\right)}$

8) $y_x := \left(\cos\left(x^2 + 23\cdot x - 76\right) + \sin\left(x^3 - 12\cdot x + 49\right)\right)\cdot e^{x - 10}$

9) $y_x := \sin(\pi - x)\cdot\ln\left(x^2 + 8\cdot x\right) + \cos\left(\sqrt{x^2 - 12}\right)\cdot e^{-x}$

10) $y_x := \frac{x^4 - 12 \cdot x^3 + 31 \cdot x}{21 \cdot x^2 - 17} - \frac{15 \cdot x^3 + 14}{x + 5}$

11) $y_x := \ln\left(\frac{x^3 - 18}{x^2 + 11}\right) - e^{\sqrt{x^2 + 26}} \cdot \cos(\pi - x)$

12) $y_x := e^{\frac{x^4 + 13 \cdot \sqrt{x - 14}}{x^5 - 12 \cdot x^2}} + 2x^{\frac{3}{x}}$

13) $y_x := \sin\left(\frac{x^2 - 17}{x^3 + 15}\right) + \frac{\cos\left(\frac{x^3 + 22}{x + 16}\right)}{\sin\left(e^x\right)}$

14) $y_x := \left(\sin\left(x - \frac{\pi}{2}\right) + \cos\left(10 \cdot x - x^2\right)\right)^2 - \ln\left(x^3 - \sqrt{3x - 12}\right)$

15) $y_x := \cos\left(\frac{\pi}{3} - x\right) \cdot \frac{5 \cdot x^3 - 22}{17 \cdot x^2 - 5} + e^{\frac{-1}{x}}$

16) $y_x := \left(5 \cdot x^4 - 7 \cdot x^3\right) \cdot \log\left(e^{-x}\right) + 12 + \cos\left(x^2 - 17\right)$

17) $y_x := \left(3 \cdot x^3 - 12 \cdot x^2 + 15\right) \cdot \left(\frac{\cos\left(x^2 - 5 \cdot x\right)}{\sin\left(\frac{\pi}{3} \cdot 5 - x\right)}\right)$

18) $y_x := \frac{-12 \cdot x^4 + 5 \cdot x^3 - 23}{e^{13\,x - x^2}} \cdot \cos\left(\log\left(x^2\right) - 15 \cdot \frac{\pi}{7}\right)$

7 РАБОТА С ФАЙЛАМИ В ЯЗЫКЕ ASSEMBLER

7.1 Создание файла

Функция DOS 3Ch — Создать файл

Ввод:	AX = 3Ch CX = атрибут файла бит 7: файл можно открывать разным процессам в Novell Netware бит 6: не используется бит 5: архивный бит (1, если файл не сохранялся) бит 4: каталог (должен быть 0 для функции 3Ch) бит 3: метка тома (игнорируется функцией 3Ch) бит 2: системный файл бит 1: скрытый файл бит 0: файл только для чтения DS:DX = адрес ASCIZ-строки с полным именем файла (ASCIZ-строка ASCII-символов, оканчивающаяся нулем)
Вывод:	CF = 0 и AX = идентификатор файла, если не произошла ошибка CF = 1 и AX = 03h, если путь не найден CF = 1 и AX = 04h, если слишком много открытых файлов CF = 1 и AX = 05h, если доступ запрещен

Если файл уже существует, функция 3Ch все равно открывает его, присваивая ему нулевую длину. Чтобы этого не произошло, следует пользоваться функцией 5Bh.

Пример:

```
mov al,00h
lea dx,name1 ; устанавливаем имя файла name1
mov ah,3ch
int 21h ; создаем файл
```

7.2 Открытие существующего файла

Ввод:	AX = 3Dh AL = режим доступа биты 0 – : права доступа 00: чтение 01: запись 10: чтение и запись бит 1: открыть для записи биты 2 – 3: зарезервированы (0) биты 6 – 4: режим доступа для других процессов 000: режим совместимости (остальные процессы также должны открывать этот файл в режиме совместимости) 001: все операции запрещены 010: запись запрещена 011: чтение запрещено 100: запрещений нет бит 7: файл не наследуется порождаемыми процессами DS:DX = адрес ASCIZ-строки с полным именем файла CL = маска атрибутов файлов
Вывод:	CF = 0 и AX = идентификатор файла, если не произошла ошибка CF = 1 и AX = код ошибки (02h — файл не найден, 03h — путь не найден, 04h — слишком много открытых файлов, 05h — доступ запрещен, 0Ch — неправильный режим доступа)

Пример:

```
mov al,02h ; открываем файл на чтение\запись
lea dx,name1 ; устанавливаем имя файла name1
mov ah,3dh
int 21h ; открываем файл
```

7.3 Создание и открытие файла.

Создать и открыть новый файл

Ввод:	AX = 5Bh CX = атрибут файла DS:DX = адрес ASCIZ-строки с полным именем файла
Вывод:	CF = 0 и AX = идентификатор файла, открытого для чтения/записи в режиме совместимости, если не произошла ошибка CF = 1 и AX = код ошибки (03h — путь не найден, 04h — слишком много открытых файлов, 05h — доступ запрещен, 50h — файл уже существует)

Функция DOS 5Ah — Создать и открыть временный файл

Ввод:	AX = 5Ah CX = атрибут файла DS:DX = адрес ASCIZ-строки с путем, оканчивающимся символом «\», и тринадцатью нулевыми байтами в конце
Вывод:	CF = 0 и AX = идентификатор файла, открытого для чтения/записи в режиме совместимости, если не произошла ошибка (в строку по адресу DS:DX дописывается имя файла) CF = 1 и AX = код ошибки (03h — путь не найден, 04h — слишком много открытых файлов, 05h — доступ запрещен)

Функция 5Ah создает файл с уникальным именем, который не является на самом деле временным, его следует специально удалять, для чего его имя и записывается в строку в DS:DX.

Во всех случаях строка с полным именем файла имеет вид типа

```
Name1 db 'c:\data\filename.ext',0
```

причем, если диск или путь опущены, используются их текущие значения.

Для работы с длинными именами файлов в DOS 7.0 (Windows 95) и старше используется еще один дополнительный набор функций, которые вызываются как функция DOS 71h.

Функция LFN 6Ch — Создать или открыть файл с длинным именем

Ввод:	AX = 716Ch BX = режим доступа Windows 95 биты 2 – 0: доступ 000 — только для чтения 001 — только для записи 010 — для чтения и записи 100 — только для чтения, не изменять время последнего обращения к файлу биты 6 – 4: доступ для других процессов (см. функцию 3Dh) бит 7: файл не наследуется порождаемыми процессами бит 8: данные не буферизуются бит 9: не архивировать файл, если используется архивирование файловой системы (DoubleSpace) бит 10: использовать число в DI для записи в конец короткого имени файла бит 13: не вызывать прерывание 24h при критических ошибках бит 14: сбрасывать буфера на диск после каждой записи в файл CX = атрибут файла DX = действие бит 0: открыть файл (ошибка, если файл не существует) бит 1: заменить файл (ошибка, если файл не существует) бит 4: создать файл (ошибка, если файл существует) DS:SI = адрес ASCIZ-строки с именем файла DI = число, которое будет записано в конце короткого варианта имени файла
Вывод:	CF = 0 AX = идентификатор файла CX = 1, если файл открыт CX = 2, если файл создан CX = 3, если файл заменен CF = 1, если произошла ошибка AX = код ошибки (7100h, если функция не поддерживается)

Если функции открытия файлов возвращают ошибку «слишком много открытых файлов» (AX = 4), следует увеличить число допустимых идентификаторов с помощью функции 67h.

7.4 Чтение, запись и переименование файла

Чтение из файла или устройства

Ввод:	AH = 3Fh BX = идентификатор CX = число байт DS:DX = адрес буфера для приема данных
Вывод:	CF = 0 и AX = число считанных байт, если не произошла ошибка CF = 1 и AX = 05h, если доступ запрещен, 06h, если неправильный идентификатор

Пример:

```
 mov bx,ax ; идентификатор файла в BX
mov cx,1 ; считывать один байт
mov dx,offset buffer ; начало буфера - в DX
mov ah,3Fh ; чтение файла
int 21h
```

Если при чтении из файла число фактически считанных байт в AX меньше, чем заказанное число в CX, при чтении был достигнут конец файла. Каждая следующая операция чтения, так же как и записи, начинается не с начала файла, а с того байта, на котором остановилась предыдущая операция чтения/записи. Если требуется считать (или записать) произвольный участок файла, используют функцию 42h (функция lseek в C).

7.5 Перемещение указателя чтения/записи

Ввод:	AH = 42h BX = идентификатор CX:DX = расстояние, на которое надо переместить указатель (со знаком) AL = перемещение от:

	0 — от начала файла 1 — от текущей позиции 2 — от конца файла
Вывод:	CF = 0 и CX:DX = новое значение указателя (в байтах от начала файла), если не произошла ошибка CF = 1 и AX = 06h, если неправильный идентификатор

Пример:

```
mov ax,4201h ; переместить указатель файла от текущей
dec cx ; позиции назад на 1
dec cx ; CX = FFFFh
mov dx,cx ; DX = FFFFh
int 21h
```

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

7.6 Запись в файл или устройство

Ввод:	AH = 40h BX = идентификатор CX = число байт DS:DX = адрес буфера с данными
Вывод:	CF = 0 и AX = число записанных байт, если не произошла ошибка CF = 1 и AX = 05h, если доступ запрещен, 06h, если неправильный идентификатор

Если при записи в файл указать CX = 0, файл будет обрезан по текущему значению указателя. При записи в файл на самом деле происходит запись в буфер DOS, данные из которого сбрасываются на диск при закрытии файла или если их

количество превышает размер сектора диска. Для немедленного сброса буфера можно использовать функцию 68h (функция fflush в C).

Пример: с переносом указателя и записью в файл

```
mov ax,4201h ; переместить указатель файла от текущей
dec cx ; позиции назад на 1
dec cx ; CX = FFFFh
mov dx,cx ; DX = FFFFh
int 21h
mov ah,40h ; записать в файл
```

7.7 Переименование файла

Ввод:	AH = 56h DS:DX = ASCIIZ- имя существующего файла ES:DI = ASCIIZ- имя нового файла CL = Маска атрибутов
Вывод:	CF = 0, если операция выполнена CF = 1, если произошла ошибка (AX = код ошибки)

Функция 56h позволяет произвести перемещение между каталогами, не изменяя устройства.

Пример: Перемещения между каталогами без смены устройства

```
.data
fname_s db ‘name1.asm’
point_fname_s dd fname_s
fname_d db ‘e:\name1.asm’
point_fname_d dd fname_d
.
.
.
.code ; переместим файл из текущего каталога в корневой
lds dx.point_fname_s ; указатель на исходный файл
les di.point_fname_d ; указатель на новый файл
mov ah,56h
int 21h ; перемещаем файл
```

7.8 Закрытие и удаление файла

7.8.1 Закрыть файл

Ввод:	AH = 3Eh BX = идентификатор
Вывод:	CF = 0, если не произошла ошибка CF = 1 и AX = 6, если неправильный идентификатор

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

7.8.2 Удаление

Функция DOS 41h — Удаление файла

Ввод:	AH = 41h DS:DX = адрес ASCIZ-строки с полным именем файла
Вывод:	CF = 0, если файл удален CF = 1 и AH = 02h, если файл не найден, 03h — если путь не найден, 05h — если доступ запрещен

Удалить файл можно только после того, как он будет закрыт, так как DOS будет продолжать выполнять запись в несуществующий файл, что может привести к разрушению файловой системы. Функция 41h не позволяет использовать маски (символы * и ? в имени файла) для удаления сразу нескольких файлов, хотя этого можно добиться, вызывая ее через недокументированную функцию 5D00h. Но, начиная с DOS 7.0 (Windows 95), официальная функция удаления файла может работать сразу с несколькими файлами.

7.9 Удаление файлов с длинным именем

Ввод:	AX = 7141h DS:DX = адрес ASCIZ-строки с длинным именем файла SI = 0000h: маски не разрешены и атрибуты в CX игнорируются SI = 0001h: маски в имени файла и атрибуты в CX разрешены: CL = атрибуты, которые файлы могут иметь CH = атрибуты, которые файлы должны иметь
Вывод:	CF = 0, если файл или файлы удалены CF = 1 и AX = код ошибки, если произошла ошибка. Код 7100h означает, что функция не поддерживается

7.10 Поиск файлов

7.10.1 Найти первый файл

Найти нужный файл на диске намного сложнее, чем просто открыть его, — для этого требуются две функции при работе с короткими именами (найти первый файл и найти следующий файл) и три — при работе с длинными именами в DOS 7.0 (найти первый файл, найти следующий файл, прекратить поиск).

Функция DOS 4Eh — Найти первый файл

Ввод:	AH = 4Eh AL используется при обращении к функции APPEND CX = атрибуты, которые должен иметь файл (биты 0 (только для чтения) и 5 (архивный бит) игнорируются, если бит 3 (метка тома) установлен, все остальные биты игнорируются) DS:DX = адрес ASCIZ-строки с именем файла, которое может включать путь и маски для поиска (символы * и ?)
Вывод:	CF = 0 и область DTA заполняется данными, если файл найден CF = 1 и AX = 02h, если файл не найден, 03h — если путь не найден, 12h — если неправильный режим доступа

Вызов этой функции заполняет данными область памяти DTA (область передачи данных), которая начинается по умолчанию со смещения 0080h от начала

блока данных PSP (при запуске COM- и EXE-программ сегменты DS и ES содержат сегментный адрес начала PSP), но ее можно переопределить с помощью функции 1Ah.

Функция DOS 1Ah — Установить область DTA

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

Функции поиска файлов заполняют DTA следующим образом:

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

+01h: 11 байт — маска поиска (без пути)

+0CH: байт — атрибуты для поиска

+0Dh: слово — порядковый номер файла в каталоге

+0Fh: слово — номер кластера начала внешнего каталога

+11h: 4 байта — зарезервировано

+15h: байт — атрибут найденного файла

+16h: слово — время создания файла в формате DOS:

биты 15 – 11: час (0 — 23)

биты 10 – 5: минута

биты 4 – 0: номер секунды, деленный на 2 (0 – 30)

+18h: слово — дата создания файла в формате DOS:

биты 15 – 9: год, начиная с 1980

биты 8 – 5: месяц

биты 4 – 0: день

+1Ah: 4 байта — размер файла

+1Eh: 13 байт — ASCIZ-имя найденного файла с расширением

После того как DTA заполнена данными, для продолжения поиска следует вызывать функцию 4Fh, пока не будет возвращена ошибка.

7.10.2 Найти следующий файл

Функция DOS 4Fh — Найти следующий файл

Ввод:	AH = 4Fh DTA — содержит данные от предыдущего вызова функции 4E или 4F
Вывод:	CF = 0 и DTA содержит данные о следующем найденном файле, если не произошла ошибка CF = 1 и AX = код ошибки, если произошла ошибка

Для случая длинных имен файлов (LFN) употребляется набор из трех подфункций функции DOS 71h, которые можно использовать, только если запущен IFSmgr (всегда запускается при обычной установке Windows 95, но не запускается, например, с загрузочной дискеты MS-DOS 7.0).

В качестве примера программы, использующей многие из функций работы с файлами, рассмотрим программу, заменяющую русские буквы «Н» на латинские «H» во всех файлах с расширением .TXT в текущем каталоге.

```
; fidoh.asm
; заменяет русские "Н" на латинские "H" во всех файлах с расширением .TXT
; в текущем каталоге
 .model tiny
 .code
 org 100h ; COM-файл
start:
 mov ah,4Eh ; поиск первого файла
 xor cx,cx ; не системный, не каталог и т.д.
 mov dx,offset filespec ; маска для поиска в DS:DX
file_open:
 int 21h
 jc no_more_files ; если CF = 1 - файлы кончились

 mov ax,3D02h ; открыть файл для чтения и записи
 mov dx,80h+1Eh ; смещение DTA + смещение имени файла
 int 21h ; от начала DTA
 jc find_next ; если файл не открылся - перейти
 ; к следующему
```

```
 mov bx,ax ; идентификатор файла в BX
 mov cx,1 ; считывать один байт
 mov dx,offset buffer ; начало буфера - в DX
read_next:
 mov ah,3Fh ; чтение файла
 int 21h
 jc find_next ; если ошибка - перейти к следующему
 dec ax ; если AX = 0 - файл кончился -
 js find_next ; перейти к следующему
 cmp byte ptr buffer,8Dh ; если не считана русская "Н",
 jne read_next ; считать следующий байт,
 mov byte ptr buffer,48h ; иначе - записать в буфер
 ; латинскую букву "H"
 mov ax,4201h ; переместить указатель файла от текущей
 dec cx ; позиции назад на 1
 dec cx ; CX = FFFFh
 mov dx,cx ; DX = FFFFh
 int 21h
 mov ah,40h ; записать в файл
 inc cx
 inc cx ; один байт (CX = 1)
 mov dx,offset buffer ; из буфера в DS:DX
 int 21h
 jmp short read_next ; считать следующий байт
find_next:
 mov ah,3Eh ; закрыть предыдущий файл
 int 21h
 mov ah,4Fh ; найти следующий файл
 mov dx,80h ; смещение DTA от начала PSP
 jmp short file_open
no_more_files: ; если файлы кончились,
 ret ; выйти из программы
filespec db "*.txt",0 ; маска для поиска
buffer label byte ; буфер для чтения/записи -
 end start ; за концом программы
```

7.11 Задания

Во всех вариантах необходимо реализовать программу работы с файлами. Пользователь вводит с клавиатуры имя файла с текстом и имя создаваемого файла, в который будет помещен результат. Слова в строке могут быть разделены пробелами и знаками препинания.

Варианты заданий:

1) Выровнять все строки в файле по правому краю
2) Выровнять все строки в файле по центру
3) Выровнять все строки в файле по левому краю
4) Отформатировать файл таким образом, чтобы длина каждой строки не превышала заданного пользователем значения
5) Отсортировать строки файла по длинам
6) Отсортировать строки файла по количеству гласных букв
7) Отсортировать строки файла по алфавиту по первым трем буквам
8) Отсортировать строки файла по количеству одинаковых букв
9) Найти в каждой строке файла заданную пользователем последовательность символов и заменить на введенное слово

СПИСОК ЛИТЕРАТУРЫ:

1) Питер Абель Assembler и программирование для IBM PC 1995.
2) Юров В. Assembler, 2001
3) Зубков С.В. Assembler. Язык неограниченных возможностей. 1999.
4) Пильщиков В.Н. Программирование на языке ассемблера IBM PC. 1997.
5) Орлов С.Б. Программа-справочник по системе программирования турбо ассемблер 2.0. Руководство пользователя. М. 1990г.