Пензенский государственный университет

Кафедра «МОиПЭВМ»

Методические указания к лабораторным работам по дисциплине «**Программирование на машинно-ориентированных языках**»

Пенза 2022

В общем виде структура программы на ассемблере masm32 состоит из сегментов следующих типов:

– сегмент кода, содержащий собственно текст программы;
– сегменты данных:
  • сегмент констант, содержащий директивы объявления данных, изменение которых в программе не предполагается;
  • сегмент инициализированных данных, содержащий директивы объявление данных, для которых заданы начальные значения – память под эти данные распределяется во время ассемблирования программы;
  • сегмент неинициализированных данных, содержащий директивы объявление данных – память под эти данные отводится во время загрузки программы на выполнение;
  • сегмент стека, определяемый для ассемблера по заданному размеру.

В программе сегменты описываются полными или сокращенными директивами. Сокращенные директивы описания сегмента кодируются следующим образом:

`.CODE [Имя сегмента]` – начало или продолжение сегмента кода;

`.MODEL Модель [Модификатор][,Язык][,Модификатор языка],`

где `Модель` – определяет набор и типы сегментов; при 32-х разрядной адресации используется единственная модель `FLAT`;

`Модификатор` – определяет тип адресации: use16, use32, dos;

`Язык и Модификатор языка` – определяют особенности передачи параметров при вызове подпрограмм на разных языках C, PASCAL, STDCALL, последняя – это гибрид C и PASCAL. Согласно ему, данные передаются справа налево, но вызываемый ответственен за очистку стека. Платформа Win32 использует исключительно `STDCALL`;

`.DATA` – начало или продолжение сегмента инициализированных данных;

`.DATA?` – начало или продолжение сегмента неинициализированных данных;

`.CONST` – начало или продолжение сегмента неизменяемых данных;

`.STACK [Размер]` – начало или продолжение сегмента стека.

В общем виде шаблон программы можно представить следующим образом.

```
.586      ; набор операций для процессора 80586
.MODEL Flat, STDCALL
.DATA
 < инициализированные данные>
.DATA?
 < неинициализированные данные>
.CONST
 < константы>
.CODE
 <метка>
 < код>
end <метка>
```

Для ввода и последующей обработки целого числа можно использовать следующий шаблон.

```
; Шаблон консольной программы (Ввод и вывод целого числа)
 .586
 .MODEL flat, stdcall
 OPTION CASEMAP:NONE     ;чувствительность к регистру
Include kernel32.inc
Include masm32.inc
IncludeLib kernel32.lib
IncludeLib masm32.lib
```

```
 .CONST
MsgExit DB 13,10,"Press Enter to Exit",0AH,0DH,0
 .DATA

Zapros DB 13,10,'Input A',13,10,0
Result DB 'Result='
ResStr DB 16 DUP (' '),0
 .DATA?
A SWORD ?

Buffer DB 10 DUP (?)
inbuf DB 100 DUP (?)
 .CODE
Start: Invoke StdOut,ADDR Zapros ;Вывод запроса
 Invoke StdIn,ADDR Buffer,LengthOf Buffer ; Ввод строки
 Invoke StripLF,ADDR Buffer ;Добавить 0 в конец строки
 Invoke atol,ADDR Buffer ;Строка в число EAX
 mov DWORD PTR A,EAX ;Запись числа по адресу A
; ---------------------
; Текст программы
; ---------------------
 Invoke dwtoa,A,ADDR ResStr ;Число в строку
 Invoke StdOut,ADDR Result ;Вывод строки
 XOR EAX,EAX
 Invoke StdOut,ADDR MsgExit ;Для завершения Enter
 Invoke StdIn,ADDR inbuf,LengthOf inbuf ;Ввод
 Invoke ExitProcess,0 ;Выход с кодом 0
 End Start
```

Ввод, преобразование и вывод осуществляются с помощью процедуры `INVOKE`. Первый параметр – название функции. Прототипы функций включаются в программу с помощью директивы `Include`. Директива `IncludeLib` включает скомпилированные тексты стандартных функций. При этом используются библиотеки `kernel32.inc`, `masm32.inc`, `kernel32.lib` и `masm32.lib`.

Для подключения отладчика Ollydbg необходимо поместить его в один из каталогов, например, в каталог, в RadAsm, где расположена среда. В пункте главного меню *Настройки* в меню выбирается пункт *Установить пути* и в окне *Debug* прописать путь до каталога, в котором находится отладчик.

Если необходимо использовать ассемблер masm32 без среды разработки RadAsm, то для создания того же шаблона или другой программы можно использовать командный файл makeit.bat.

```
@echo off
c:\masm32\bin\ml /c /coff /Cp /nologo /I"C:\Masm32\Include"
%1.asm
if errorlevel 1 goto errasm

c:\masm32\bin\Link /SUBSYSTEM:CONSOLE /RELEASE
/LIBPATH:"C:\Masm32\Lib" /OUT:"%1.exe" %1.obj
if errorlevel 1 goto errlink
goto TheEnd
:errlink
echo _
```

```
echo Link error
goto TheEnd
:errasm
echo _
echo Assembly Error
goto TheEnd
:TheEnd
pause
```

Ключи при обращении к ассемблеру `c:\masm32\bin\ml` имеют следующее назначение:

`/c` – заказывает ассемблирование без автоматической компоновки,

`/coff` – определяет формат объектного модуля Microsoft (coff),

`/Cp` – означает сохранение регистра строчных и прописных букв всех идентификаторов программы,

`/nologo` – осуществляет подавление вывода сообщений на экран в случае успешного завершения ассемблирования,

`/I"C:\Masm32\Include"` – определяет местонахождение вставляемых (.inc) файлов,

параметр `"%1.asm"` – задает имя обрабатываемого файла.

Ключи при вызове компоновщика `c:\masm32\bin\Link:`

`/SUBSYSTEM:CONSOLE` – подключить стандартное окно консоли,

`/RELEASE` – создать реализацию (а не отладочный вариант),

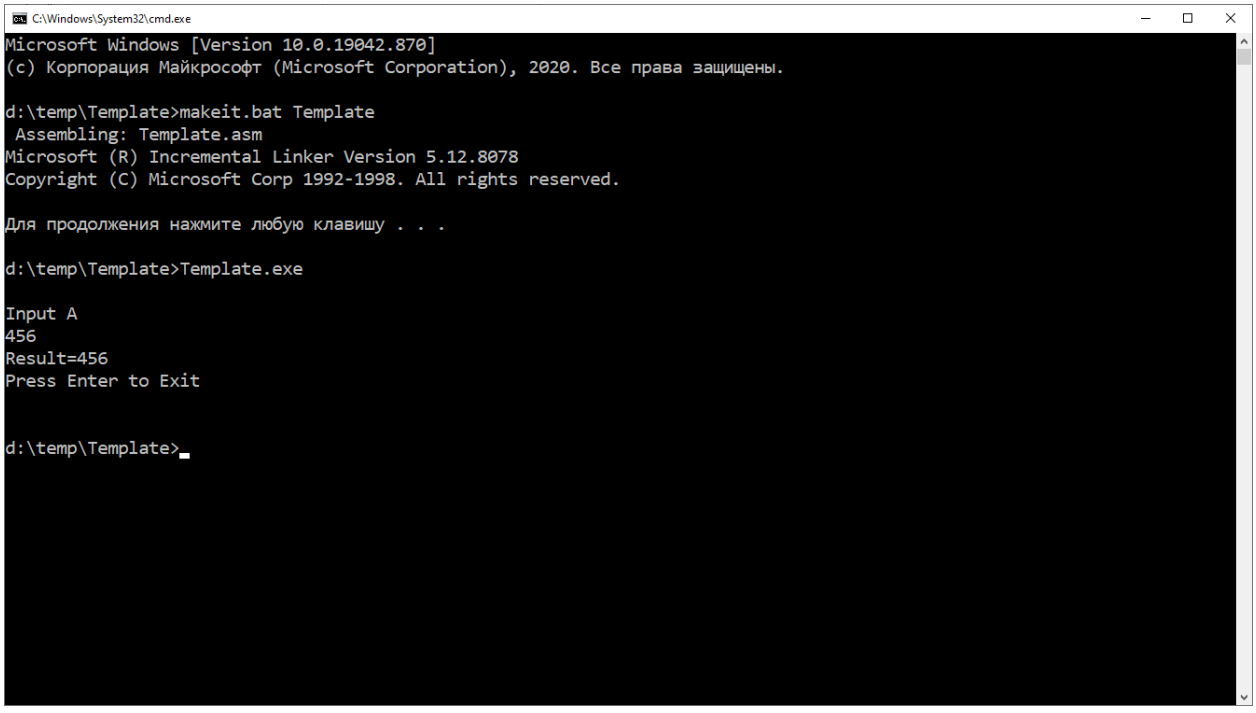
`/VERSION:4.0` – минимальная версия компоновщика,

`/LIBPATH:"C:\Masm32\Lib"` – путь к файлам библиотек,

`/OUT:"%1.exe"` – имя результата компоновки – загрузочного файла,

параметр %1.obj определяет имя объектного файла, полученного ассемблером.

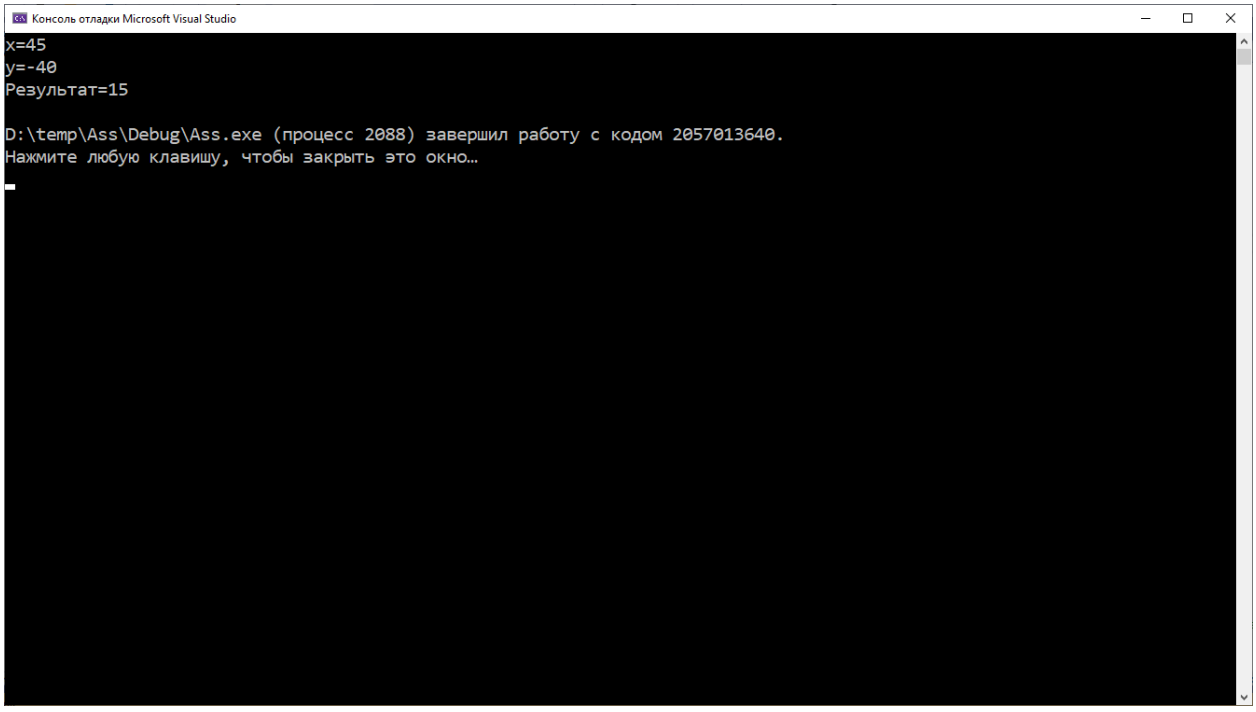
При запуске этого командного файла получается следующее:



При использовании в качестве среды разработки MS Visual Studio можно применять в тексте программы на C++ ассемблерные вставки. Пример программы с ассемблерной вставкой.

```
#include <iostream>
using namespace std;
int main()
{
	setlocale(LC_ALL, "");
	int x, y, result;
	cout << "x=";  cin >> x;
	cout << "y=";  cin >> y;
	__asm
	{
		mov eax, x
		add eax, y
		add eax, 10
		mov result, eax
	}
	cout << "Результат="<< result<< endl;
}
```

Результат выполнения программы:



Лабораторная работа № 1. Программирование арифметических выражений

Цель работы: научиться программировать арифметические выражения.

Процессоры семейства IA-32 поддерживают арифметические операции над однобайтовыми, двухбайтовыми и четырехбайтовыми целыми числами.

Размер операндов при этом определяется:

- объемом регистра, хранящего число – если хотя бы один операнд находится в регистре;
- размером числа, заданным директивой определения данных;
- специальными описателями, например, BYTE PTR (байт), WORD PTR (слово) и DWORD PTR (двойное слово), если ни один операнд не находится в регистре и размер операнда отличен от размера, указанного директивой определения данных.

Команда INC – прибавление 1.

```
INC <операнд>
```

Команда INC прибавляет 1 к операнду, не изменяя состояние бита переноса CF в регистре признаков. Операндом может быть регистр или ячейка памяти. Признаки OF, SF, ZF, AF, PF устанавливаются в соответствии с результатом.

Пример:

```
inc ax ; ax=ax+1;
```

Команда DEC – вычитание 1

```
DEC <операнд>
```

Команда DEC вычитает 1 из операнда, не изменяя состояние бита переноса CF в регистре признаков. Операндом может быть регистр или ячейка памяти. Признаки OF, SF, ZF, AF, PF устанавливаются в соответствии с результатом.

Пример:

```
dec ax ; ax=ax-1
```

Команда ADD – сложение целых чисел

```
ADD <приемник>,<источник>
```

Команда ADD прибавляет операнд источника к операнду приемника и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Может оперировать с двумя знаковыми или беззнаковыми целыми числами разрядностью 8, 16 или 32 бита. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом. Если установлены признаки OF и CF, то результат операции вышел за разрядную сетку операнда.

Примеры:

```
add ax,1 ; ax=ax+1
add ax,bx ; ax=ax+bx
```

Команда SUB – вычитание целых чисел

```
SUB <приемник>,<источник>
```

Команда SUB вычитает операнд источника из операнда приемника и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Может оперировать с двумя знаковыми или беззнаковыми целыми числами разрядностью 8, 16 или 32 бита. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, SF, ZF, AF, PF, CF устанавливаются в соответствии с результатом. Если установлены признаки OF и CF, то результат операции вышел за разрядную сетку операнда. Флаг SF отображает знак результата.

Примеры:

```
sub ax,2 ; ax=ax-2
sub ax,bx ; ax=ax-bx
```

Команда IMUL/MUL – умножение целых чисел

```
IMUL <множитель>
MUL <множитель>
```

Команда IMUL производит умножение двух чисел со знаком. Команда MUL производит умножение двух чисел без знака. Операнд <множитель>может быть регистром, ячейкой памяти или числом. Второй операнд содержится в регистре AL или AX (в зависимости от формата операнда источника). Результат умножения сохраняется в регистре AX или в паре DX:AX соответственно. Признаки OF, CF устанавливаются в случае, если старший бит (включая знаковый бит) переходит в старший регистр результата. Значения битов SF, ZF, AF, PF не определены.

Пример:

```
mov al,2
mul 4 ; ax=8
```

Команда IDIV/DIV – деление целых чисел

```
IDIV <делитель>
DIV <делитель>
```

Команда IDIV производит деление со знаком. Команда DIV производит деление без знака. Операнд <делитель> может быть регистром, ячейкой памяти или числом. Делимое содержится в регистре AX или паре DX:AX (в зависимости от формата операнда источника). Результат деления сохраняется в регистре AL или AX, а остаток – в регистрах AH или DX соответственно. Знак частного всегда совпадает со знаком делимого. Абсолютная величина частного всегда меньше абсолютной величины делимого. Значения битов OF, SF, ZF, AF, PF, CF не определены.

Пример:

```
mov ax,21
div 4 ; al=5, ah=1
```

Команда CBW – преобразование байта в слово

Команда CBW удваивает размер операнда, содержащегося в регистре AL (используется по умолчанию). Копирует знаковый бит (бит 15) регистра AL во все биты регистра AH.

Пример:

```
mov al,-21h ; ah=?, al=11011111b
cbw ; ah=11111111b, al=11011111b
```

Команда CWD – преобразование слова в двойное слово

Команда CWD удваивает размер операнда, содержащегося в регистре AX (используется по умолчанию). Копирует знаковый бит (бит 31) регистра AX во все биты регистра DX.

Пример:

```
mov ax,-1234h ; ah=11101101h, al=11001100b
cwd ; dx=11111111 11111111b
```

Команда SAR/SAL/SHR/SHL – арифметический сдвиг вправо/влево, логический сдвиг вправо/влево

```
SAR <приемник>,<число_разрядов>
SAL <приемник>,<число_разрядов>
SHR <приемник>,<число_разрядов>
SHL <приемник>,<число_разрядов>
```

Команды SAR/SAL/SHR/SHL сдвигает биты в операнде <приемник> на указанное <число_разрядов> вправо/влево. Каждый сдвигаемый бит помещается в признак CF и при следующем сдвиге бита опускается. Операнд <приемник> может быть регистром или ячейкой памяти. Число сдвигаемых разрядов варьируется от 1 до 5. Если число сдвигаемых разрядов не указано, то производится сдвиг на 1 разряд.

Команды SAL/SHL производят одинаковые действия. Старший бит сдвигается в признак переноса CF, а младший бит сбрасывается.

Команды SAR/SHR производят сдвиг вправо. Младший бит сдвигается в признак переноса CF, а старший бит сбрасывается командой SHR или устанавливается равным предыдущему значению (знаку числа) командой SAR.

Таким образом, команда SHR может применяться для беззнакового деления на 2, 4, 8, 16, 32, а команда SAR – для знакового деления. При сдвиге влево признак OF сбрасывается, если признак CF равен старшему биту результату, устанавливается в 1 в противном случае. При сдвиге вправо использование команды SHR устанавливает бит OF при сдвиге каждого бита. При использовании команды SAR признак OF сбрасывается. Признак OF изменяется только при сдвиге на 1 бит.

Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда <приемник>. Признаки SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,11001101b
sar al,2 ; al=11110011b
shr al,1 ; al=01111001b
```

Команда ROR/ROL/RCR/RCL – циклический сдвиг вправо/влево, циклический сдвиг вправо/влево через перенос

```
ROR <приемник>,<число_разрядов>
ROL <приемник>,<число_разрядов>
RCR <приемник>,<число_разрядов>
RCL <приемник>,<число_разрядов>
```

Команды ROR/ROL/RCR/RCL циклически сдвигает биты в операнде <приемник> на указанное <число_разрядов> вправо/влево.

Операнд приемника может быть регистром или ячейкой памяти. Число сдвигаемых разрядов варьируется от 1 до 5. Признак CF устанавливается равным последнему сдвигаемому биту. Признак OF изменяется только при сдвиге на 1 бит. Признаки SF, ZF, AF и PF не изменяются.

Примеры:

```
mov al,11001101b
rol al,2 ; al=00110111b
rcr al,2 ; al=11001101b CF=1
```

Возможности процессора по вычислению исполнительного адреса можно задействовать и отдельно от обращения к памяти. Для этого предусмотрена команда lea. Команда имеет два операнда, причём первый из них обязан быть регистровым (размером 2 или 4 байта), а второй операндом типа "память" При этом никакого обращения к памяти команда не делает вместо этого в регистр, указанный первым операндом, заносится адрес, вычисленный обычным способом для второго операнда. Если первый операнд двухбайтный регистр, то в него будут записаны младшие 16 бит вычисленного адреса. Например, команда

```
lea eax,[1000+ebx+8*ecx]
```

возьмет значение регистра ECX, умножит его на 8, прибавит к этому значение регистра EBX и число 1000, а полученный результат занесет в регистр EAX. Разумеется, вместо числа можно использовать и метку. Ограничения на выражение в скобках точно такие же, как и в других случаях использования операнда типа "память".

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Варианты заданий

Номер варианта	Выражение	Номер варианта	Выражение
1	X=a+2*b-c*d	2	X=a-c/d+b*4
3	X=b*(c-d)+a/2	4	X=(a+b)/(a-2*b)
5	X=5*a+c*b	6	X=7*a/(b-c)
7	X=(a+b-c)*(d-1)	8	X=a/b+2*c/d
9	X=a+b*c-d/4	10	X=a+2*c/(b-1)
11	X=a/2+b*c*d	12	X=8*a/(b-c)+d
13	X=4*a+12*(b-c+1)	14	X=3*a+b/(c+2)
15	$X=a^2+b^3-1$	16	$X=a+b^2/c-d/2$
17	X=a*b/2+c-1	18	$X=a^2+(b+c)/d$
19	X=a*b*c-d/4	20	X=(a-b)/(2*c+1)
21	$X=a/4+b*c^2$	22	X=a/b+c-d
23	$X=a+(b^2-c^2)/2$	24	X=a+2*b/(c+d)
25	X=a*b-c/2+d*3	26	X=144-a/(b+c)
27	X=a+b*c*(d-3)	28	X=1024/a+b*8-1
29	X=(a+b-c)*(b+d-1)	30	X=4*a-(b+c)/d

Лабораторная работа № 2. Разветвляющиеся алгоритмы

Цель работы: научиться программировать алгоритмы с условными операторами.

Команда `JMP` – безусловный переход.

```
JMP адрес
```

Команда `JMP` осуществляет передачу программного управления в другую точку кода, не запоминая информацию для возврата. Операнд адрес определяет адрес команды, которой должно быть передано управление. Операндом в данной команде может быть число, регистр общего назначения или ячейка памяти. Команда `JMP` может применяться для осуществления следующих четырех типов переходов:

- `near` – переход к команде в том же сегменте кода (физический адрес сегмента кода содержится в регистре CS);
- `short` – переход типа near к команде, находящейся в адресном пространстве -128 …+127 байт от текущего положения программного счетчика IP;
- `far` – переход к команде, находящейся в другом сегменте кода, который имеет такой же уровень приоритета, как и текущий сегмент;
- `task` – переход к команде, находящейся в другой задаче.

Последний тип может использоваться только в защищенном режиме.

При переходах типа `near` и `short` операнд адрес задается в виде абсолютного (смещение от начала сегмента кода) или относительного (смещение от текущего счетчика команд `IP`) смещения. При этом значение регистра `CS` не изменяется.

При переходах типа `far` операнд адрес задается в виде `сегмент:смещение`, в регистр `CS` записывается физический адрес нового сегмента кода, а в регистр IP – адрес смещения в новом сегменте кода.

Команды условного перехода используются после команд сравнения и арифметических команд. Для принятия решения о том, осуществлять или нет переход,

команды перехода анализируют различные комбинации флагов флажкового регистра, установленные при выполнении предыдущих команд.

Формат любой команды условного перехода выглядит следующим образом:

Мнемоника наиболее используемых команд условного перехода:

`JZ` – переход по «ноль» – `ZF=1`;

`JE` – переход по «равно» – `ZF=1`;

`JNZ` – переход по «не нуль» – `ZF=0`;

`JNE` – переход по «не равно» – `ZF=0`;

`JL` – переход по «меньше» – `SF=1`;

`JNG, JLE` – переход по «меньше или равно» – `SF=1` или `ZF=1`;

`JG` – переход по «больше» – `SF=0`;

`JNL, JGE` – переход по «больше или равно» – `SF=0` или `ZF=1`;

`JA` – переход по «выше» (беззнаковое «больше»);

`JNA, JBE` – переход по «не выше» (беззнаковое «не больше»);

`JB` – переход по «ниже» (беззнаковое «меньше»);

`JNB, JAE` – переход по «не ниже» (беззнаковое «не меньше»).

Все команды имеют однобайтовое поле адреса (формат `short`), следовательно смешение имени перехода относительно команды не должно превышать -128...127 байт. Если смещение выходит за указанные пределы, то используется специальный прием: вместо `jz zero` программируется:

```
          jnz continue
          jmp zero
continue: ...
```

Если флаг нуля установлен (`ZF=1`), то мы пропускаем условный переход и выполняем безусловный, а если сброшен, то выполняем условный переход, обходя безусловный.

Пример. Написать фрагмент вычисления X=max(A,B):

```
      mov ax,A
      cmp ax,B ; сравнение A и B
      jl LESS ; переход по меньше
      mov X,ax
      jmp CONTINUE ; переход на конец ветвления
LESS: mov ax, B
      mov X,ax
CONTINUE: ...
```

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Варианты заданий

1. Даны три числа. Возвести в квадрат те из них, значения которых неотрицательны.
2. Упорядочить по возрастанию три числа.
3. С клавиатуры вводится число, определить является ли число трехзначным.
4. Среди чисел A, B, C найти пару с наибольшей суммой.
5. Даны три числа. Выбрать из них те, которые принадлежат интервалу от 1 до 10.
6. Найти максимум из трех чисел.
7. Даны переменные A, B, C, D. Определить встречается ли переменная D. Среди остальных переменных.
8. Определить будет ли введенное число кратным 3.

9. Даны числа a, b, c. Проверить выполняется ли неравенство a<b<c.
10. Даны два числа. Вывести первое число, если оно больше второго и оба числа, если это не так.
11. Даны два неравных числа. Меньшее из них заменить полусуммой, а большее – их удвоенным произведением.
12. Вывести все цифры трехзначного десятичного числа.
13. Даны три числа. Вывести порядковый номер наименьшего из них.
14. Даны три числа. Вывести количество положительных из них.
15. Даны три числа. Найти сумму двух наибольших из них.
16. Даны три числа. Два из них равны между собой. Определить номер отличающегося числа.
17. Даны три числа. Определить какое число лежит между наибольшим и наименьшим.
18. Даны два числа. Если их значения не равны, то присвоить каждой переменной значение наибольшего числа, в противном случае обе переменных становятся равными 0.
19. Даны два числа. Проверить истинность высказывания, что оба числа четные.
20. Проверить является ли число положительным четным.

Лабораторная работа № 3. Циклы

Цель работы: научиться организовывать простые циклы.

Рассмотрим несколько возможных вариантов организации цикла с известным числом повторений. В качестве примера реализуем алгоритм вычисления суммы четных чисел от 2 до 10. В общем виде можно алгоритм можно представить следующим образом.

```
int x, sum;
x = 2;
sum = 0;
while (x<=10)
{
     sum = sum + x;
     x = x + 2;
}
```

На ассемблере этот алгоритм реализуется следующим образом:

```
     mov eax,0
     mov ebx,2
lp: cmp ebx,10
     jg cont
     add eax,ebx
     inc ebx
     inc ebx
     jmp lp
cont: mov sum,eax
```

Для обеспечения цикла while

```
while (условие)
команды_тела_цикла
```

в Ассемблере можно использовать конструкцию

```
метка:
     CMP операнд1,операнд2 ; проверка условия
     Jxx метка_выхода
          ;команды_тела_цикла
     JMP метка
```

```
метка_выхода:
```

Для обеспечения цикла do-while

```
do
команды_тела_цикла
while (условие)
```

в Ассемблере можно использовать конструкцию

```
метка:
        ;команды_тела_цикла
    CMP операнд1,операнд2
    Jxx метка
```

Для организации цикла со счетчиком удобно использовать команду LOOP <метка>. В регистр ECX нужно поместить количество повторений цикла перед командами, составляющими тело цикла. Команда работает так: значение регистра ECX уменьшается на 1, сравнивается с нулем, и, если оно не равно нулю, производится передача управления на команду, помеченную меткой. Иначе происходит переход к следующей за LOOP команде.

Пример. Найти сумму 1+2+3+…+x

```
    mov eax,0 ; сумма
    mov ecx,x ; счетчик
beg:
    add eax,ecx
    loop beg
    mov sum,eax
```

Команда LOOPZ позволяет организовать цикл с проверкой дополнительного условия. Например, мы можем уточнить условие из предыдущего примера: цикл нужно выполнить, как и раньше, не более 10 раз, но только при условии, что регистр BX содержит значение 3. Как только значение в регистре BX изменится, цикл нужно прервать.

```
    LOOPZ метка
    LOOPNZ метка
```

Команда LOOPZ уточняет условие перехода следующим образом: переход на указанную метку произойдет, если CX не содержит нуля и в то же время флаг ZF равен единице. Другое имя этой команды — LOOPE.

Следующий фрагмент кода показывает пример цикла с дополнительным условием:

```
for_start: mov CX,10 ;CX = 10
for_loop: метка для возврата назад
          тело цикла FOR
          где-то здесь изменяется регистр BX
          cmp bx,3 ;BX равен 3?
          loopz for_loop ;CX=CX-1; если CX<>0, и если BX=3 ,
                         ; переход к for_loop
for finish:              ;если CX = 0 или если BX = 3, выход
```

Команда LOOPNZ работает аналогично, но дополнительное условие противоположно: переход будет выполнен только если CX (ECX) не равен 0 и в то же время ZF равен 0. Другое имя этой команды — LOOPNE.

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Варианты заданий

1. Программа вычисления факториала целого числа.
2. Сформировать таблицу квадратов чисел от 10 до 20.
3. Найти сумму целых положительных чисел кратных 4 и меньших 30.

4. Найти произведение нечетных чисел в диапазоне от 0 до 10.
5. Дано целое число m. Получить наибольшее k, при котором 2*k меньше m.
6. Найти сумму четных чисел, лежащих в диапазоне от 2 до 16.
7. Определить количество кратных 7 чисел в диапазоне от 0 до 100.
8. Ввести с клавиатуры число N и вывести первые N чисел Фибоначчи.
9. Задать первый член арифметической прогрессии и ее разность. Вывести 10 элементов этой прогрессии.
10. Получить 8 членов геометрической прогрессии при заданных первом члене и знаменателе прогрессии.
11. Даны два целых числа, вывести все числа, лежащие в этом диапазоне и их количество.
12. Даны два целых числа N и K. Вывести N K раз.
13. Даны два числа A и N. Используя один цикл найти сумму $1+A+A^2+A^3+\ldots+A^N$.
14. Даны целые положительные числа N и K. Не используя операцию деления найти частное и остаток от деления N на K.
15. Дано целое положительное число N. Найти наименьшее целое число K, квадрат которого превосходит N.
16. Дано целое положительное число N (N>1). Найти наименьшее целое число K, при котором выполняется неравенство $3^K>N$.
17. Дано целое положительное число N (N>1). Найти наименьшее из целых чисел, для которых сумма 1+2+3+…+ K будет больше или равно N, и саму эту сумму.
18. Дано целое положительное число N (N>0). Используя операцию деления нацело найти количество и сумму его цифр.
19. Дано целое положительное число N (N>0). С помощью операции деления нацело определить имеются ли в этом числе четные цифры.
20. Проверить является ли целое число N числом Фибоначчи.

Лабораторная работа № 4. Работа с массивами

Цель работы: научиться работать с массивами.

Для обращения к элементам массива (в цикле) необходимо иметь возможность загрузить в регистр процессора адрес начала массива. Для этого используются команды `mov` или `lea`.

Команда `mov` записывается в формате

```
mov esi,OFFSET string1
```

В приведенном примере `OFFSET` – оператор ассемблера для определения адреса переменной `string1`, выполняющаяся на этапе работы ассемблера.

Команда `lea` записывается в виде

```
lea esi,string1
```

и загружает в регистр `esi` адрес переменной `string1` на этапе выполнения программы. Естественно, результаты выполнения команд `mov` и `lea` одинаковы.

«Массив данных» - это данные одинакового формата, размещенные в памяти последовательно. Массив из 5-ти однобайтных чисел, где Massl - символический адрес начала массива, то есть первого элемента массива

```
Massl db 1,-2,2,3,-3
```

Резервирование памяти для массива из 5-ти двухбайтных значений

```
Mass2 dw 5dup(?)
```

Способы адресации данных в памяти.

В командах процессора возможно несколько способов записи внутрисегментного адреса данных в памяти:

– Прямой адрес с индексированием.
– Косвенный адрес.
– Базово-индексная адресация.

Прямая адресация с индексированием

Для указания адреса элемента массива используется адрес начала массива и смещение элемента от начала массива в байтах («индекс»).

Смещение должно находиться в одном из регистров-указателей адреса - это регистры `SI, D1, BX` или `BP`, Например: `mass[si], data[di+4], data[bx-1]`.

Процессор для вычисления адреса `data[bx-1]` сложит прямой адрес `data`, значение из регистра `bx` и число -1.

Пример. Определим количество нулевых значений в массиве данных.

Используем прямую адресацию с индексированием. Реализуем циклический алгоритм, изменяя значение смещения в регистре.

Размещение данных в памяти: `mass` - массив из 8-ми двухбайтных значений в сегменте данных. Использование регистров: `si` - смещение элемента массива от его начала в байтах, `bl` - счетчик нулей, `cx` - счетчик циклов.

```
#include <iostream>
using namespace std;

int main()
{
  setlocale(LC_ALL, "r");
  int mass[8],n;
  int i;
  cout << "Ввести массив\n";
  for (i = 0; i < 8; i++)
  {
    cout << "mass[" << i << "]="; cin >> mass[i];
  }
  _asm
  {
    mov ebx,0; счетчик нулей
    mov esi,0; смещение первого элемента от mass
    mov ecx,8; счетчик циклов на 8
   cycle: cmp mass[esi],0
    jnz nxt
    inc ebx
  nxt: add esi,2; смещение до следующего элемента
    loop cycle
    mov n,ebx
  }
  cout << "n=" << n;
}
```

Косвенная адресация

Позволяет полностью уйти от использования прямого адреса в команде. Внутрисегментный адрес целиком находится в регистре-указателе адреса.

– Для использования косвенной адресации надо:

– Предварительно занести в регистр-указатель адреса - SI, DI, BX или BP - адрес начала массива. Если адрес символический, для этого предназначена команда LEA. Если адрес числовой, то команда MOV.

```
mov si,4   ; занесение числового адреса
lea si,mass ; занесение символического адреса
```

– Использовать в команде только ссылку на регистр адреса. Например, ds:[si], К регистру адреса можно добавить числовое смещение.

Например, ds:[si+2], es:[bx-20],

– В командах с операндами «косвенный адрес памяти непосредственный операнд» в записи косвенного адреса обязательно задавать для транслятора атрибут длины операнда. Например, byte ptr ds:[si], word ptr ds:[si+2].

Пример. Определим количество нулевых значений в массиве данных.

Используем косвенную адресацию элементов массива через регистр si

```
#include <iostream>
using namespace std;

int main()
{
  setlocale(LC_ALL, "");
  int mass[8], n;
  int i;
  cout << "Ввести массив\n";
  for (i = 0; i < 8; i++)
  {
    cout << "mass[" << i << "]="; cin >> mass[i];
  }
  _asm
  {
    mov ebx,0; счетчик нулей
    lea esi,mass
    mov cx,8; счетчик циклов на 8
   cycle: cmp [esi],0
    jnz nxt
    inc ebx
   nxt: add esi,4; смещение до следующего элемента
    loop cycle
    mov n, ebx
  }
  cout << "n=" << n;
}
```

Использование указателя сегмента данных по умолчанию.

Если при обращении к данным в памяти в команде не задавать регистр - указатель сегмента, процессор при выполнении команды руководствуется правилом «умолчания».

По умолчанию, процессор всегда использует регистр-указатель сегмента ds, кроме косвенной адресации через bp - в этом случае регистр ss.

```
mov address,bh ; ds:address bh
mov [si],bh   ; ds:si bh
mov [bp+4],bh  ; ss:bp+4 bh
```

Базово-индексная адресация

При базово-индексном режиме адресации для вычисления адреса операнда в памяти процессор складывает значения двух регистров, один из которых называется базовым, а другой – индексным. Для организации этого режима адресации может использоваться пара любых 32-х разрядных регистров общего назначения.

Для такого обращения к памяти при рассмотрении двумерного массива удобно использовать два адресных регистра. Разрешенные в системе команд сочетания пар регистров-указателей адреса:

```
[bx+si+disp]
[bx+di+disp]     disp - допустимая константа
[bp+si+disp]
[bp+si+disp]
```

Например,

`ds:mass[di-4]`; внутрисегментный адрес = `mass+di-4`

`ds:[di+bx]` ; внутрисегментный адрес = `di+bx`

`ds:mass[di+bx+l]` ; внутрисегментный адрес = `mass+di+bx+1`

Пример. Использование базово-индексного метода адресации.

```
.data
array word 100, 200, 300
.code
     mov ebx,offset array
     mov esi,2
     mov ax,[ebx+esi] ; ax=200

     mov edi,offset array
     mov ecx,4
     mov ax,[edi+ecx] ; ax=300

     mov ebp,offset array
     mov esi,0
     mov ax,[ebp+esi] ; ax=100
```

При использовании базово-индексной адресации для доступа к двумерным массивам (таблицам) в базовый регистр необходимо загрузить адрес строки, а в индексный регистр – смещение элемента в текущей строке.

Например, используется таблица из 3 строк и 5 столбцов

```
Table byte 10,20,0, 40,50
   byte 60,70,80, 0,100
   byte 110,120,130,140,150
numcols = 5
```

В регистр `ebx` загрузим смещение первой строки относительно начала таблицы, в регистр `esi` – номер столбца:

```
     mov ebx,numcols ; смещение строки – в данном примере равно
                ;кол-ву столбцов т к один элемент занимает 1 байт
     mov esi,2
     mov al,table[ebx+esi]
```

Пример. В матрице размером 3 «строки» на 4 «столбца» определить количество нулевых элементов.

Размещение исходных данных:

Использование регистров:

`bx`- адрес начала массива,

`si` - смещение элемента,

cx - счетчик циклов (на 12),

dx - счетчик количества нулевых слов.

```
#include <iostream>
using namespace std;

int main()
{
     int Matr[3][4] = { 1, 5, 2, 0 , 2, 3, 0, 4 , 0, 1, 0, 34 } ;
      short int n;
     _asm
     {
          mov dx,0       ;счетчик нулей
          xor eax,eax    ;eax=0
          mov cx, 12     ;счетчик цикла
          lea ebx,Matr   ;ebx=адрес массива
          mov esi,0      ; смещение первого элемента в строке
     met1: cmp [ebx][esi*4],eax
          jnz met2
          inc dx         ; +1 к счетчику нулей
     met2: add esi,1     ;смещение следующего элемента в строке
          loop met1
          mov n,dx       ;результат

     }
     std::cout << n << endl;
}
```

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Варианты заданий

1. В массиве целых положительных чисел найти наибольшее число.
2. В массиве целых чисел найти количество положительных и отрицательных чисел.
3. В массиве целых положительных чисел найти количество четных чисел не используя операцию деления.
4. Из массива целых положительных чисел создать новый массив, в котором каждое четное число уменьшено в два раза без использования операции деления.
5. Из массива целых положительных чисел создать новый массив, в котором числа расположены в обратном порядке.
6. Из массива целых чисел создать два новых массива, в один поместить положительные числа, в другой – отрицательные.
7. Даны два массива целых чисел одинакового размера. Создать новый массив и поместить в него попарные суммы элементов исходных массивов.
8. Из массива целых положительных чисел создать новый массив, в котором каждое четное число увеличено в 10 раз.
9. Создать массив, содержащий 15 первых чисел Фибоначчи: - очередное число равно сумме двух предыдущих.
10. Поменять местами соседние элементы массива.
11. В целочисленном массиве найти разности соседних элементов.

12. В заданном целочисленном массиве найти индексы тех элементов, значения которых больше заданного числа K.
13. В заданном целочисленном массиве найти номер первого отрицательного элемента делящегося на 5 с остатком 2.
14. В целочисленном массиве заменить первые K элементов на противоположные по знаку.
15. Даны массивы X=(x_1, x_2, x_3, …, x_6) и Y=(y_1, y_2, y3, … y_6). Сформировать массив Z=($z_1=x_1+y_6$, $z_2=x_2+y_5$, $z3=x_3+y_4$, … $z_6=x_6+y_1$).
16. В целочисленном массиве найти сумму отрицательных элементов.
17. Вывести все четные числа массива и количество этих чисел.
18. Дан массив целых положительных чисел. Вывести те элементы массива, которые больше своего правого соседа и количество этих чисел.
19. Проверить является ли массив упорядоченным и вывести в этом случае 1, в противном случае вывести 0.
20. Если массив целых положительных чисел является убывающим, то вывести 0. Иначе вывести номер первого элемента, нарушающего упорядоченность.

Лабораторная работа № 5. Работа со строками

Цель работы: научиться обрабатывать строки.

Символьные строки представляют собой набор символов для вывода на экран.

Для упрощения выполнения действий над массивами (непрерывными областями памяти) имеется несколько команд, объединяемых в категорию строковых операций. Именно эти команды используют регистры `ESI` и `EDI` в их особой роли.

Самые простые из строковых команд – команды `stosb`, `stosw` и `stosd`, которые записывают в память по адресу `[edi]`, соответственно, байт, слово или двойное слово из регистра AL, `AX` или `EAX`, после чего увеличивают или уменьшают (в зависимости от значения `DF`) регистр `EDI`, на 1, 2 или 4. Например, если у нас есть массив

```
buf resb 1024
```

и нам нужно заполнить его нулями, мы можем применить следующий код:

```
    xor al,al ; al=0
    lea edi,buf ; адрес начала массива
    mov ecx,1024 ; длина массива
    cld ; работа в прямом направлении
lp: stosb ; al - > [edi] , увел. edi
    loop lp
```

Эти и другие строковые команды удобно использовать с префиксом `rep`. Команда, снабженная таким префиксом, будет выполнена столько раз, какое число было в регистре `ECX` (кроме команды stosw: если её снабдить префиксом, то будет использоваться регистр `CX`; это обусловлено историческими причинами). С помощью префикса `rep` мы можем переписать вышеприведённый пример без использования метки:

```
    xor al,al
    lea edi,buf
    mov ecx,1024
    cld
    rep stosb
```

Команды `lodsb`, `lodsw` и `lodsd`, наоборот, считывают байт, слово или двойное слово из памяти по адресу, находящемуся в регистре `ESI`, и помещают прочитанное в регистр `AL`, `AX` или `EAX`, после чего увеличивают или уменьшают значение регистра `ESI` на 1, 2 или 4. Использование этих команд с префиксом `rep` обычно бессмысленно, поскольку мы не сможем между последовательными исполнениями строковой команды

вставить какие-то ещё действия, обрабатывающие значение, прочитанное и помещённое в регистр. Однако использование команд серии lods без префикса может оказаться весьма полезным. Пусть, например, у нас есть массив четырёхбайтных чисел

```
array resd 256
```

и нам необходимо сосчитать сумму его элементов. Это можно сделать следующим образом:

```
     xor ebx,ebx ; сумма=0
     lea esi,array
     mov ecx,256
     cld
lp: lodsd
     add ebx,eax
     loop lp
```

Часто оказывается удобным сочетание команд серии lods с соответствующими командами stos. Пусть, например, нам нужно увеличить на единицу все элементы того же самого массива. Это можно сделать так:

```
     lea esi,array
     mov edi,esi
     mov ecx,256
     cld
lp: lodsd
     inc eax
     stosd
     loop lp
```

Если же необходимо просто скопировать данные из одной области памяти в другую, очень удобны оказываются команды movsb, movsw и movsd. Эти команды копируют байт, слово или двойное слово из памяти по адресу [esi] в память по адресу [edi], после чего увеличивают (или уменьшают) сразу оба регистра ESI и EDI (соответственно, на 1, 2 или 4. Например, если у нас есть два строковых массива

```
buf1 resb 1024
buf2 resb 1024
```

и нужно скопировать содержимое одного из них в другой, можно сделать это так:

```
mov ecx, n
lea esi, buf1
lea edi, buf2
cld
rep movsb
```

Кроме перечисленных, процессор реализует команды cmpsb, cmpsw и cmpsd (compare string), а также scasb, scasw и scasd (scan string). Команды серии scas сравнивают аккумулятор (соответственно, AL, AX или EAX) с байтом, словом или двойным словом по адресу [edi], устанавливая соответствующие флаги подобно команде cmp, и увеличивают/уменьшают EDI. Команды серии cmps сравнивают байты, слова или двойные слова, находящиеся в памяти по адресам [esi] и [edi], устанавливают флаги и увеличивают/уменьшают оба регистра.

Кроме префикса rep, можно воспользоваться также префиксами repz и repnz (также называемыми repe и repne), которые, кроме уменьшения и проверки регистра ECX (или CX, если команда двухбайтная) также проверяют значение флага ZF и продолжают работу, только если этот флаг установлен (repz/repe) или сброшен (repnz/repne). Обычно эти префиксы используют как раз в сочетании с командами серий scas и cmps.

Длина строки

```
xor ecx,ecx ;сбрасываем ECX (ECX=0)
```

```
xor eax,eax ;EAX = 0
dec ecx ;ECX = ECX - 1 максимальная длина строки
cld      ;DF = 0, перемещение назад
repne scasb  ;поиск нулевого байта
neg ecx    ;количество выполненных итераций отрицание ECX
```

Лабораторные задания

Все данные вводятся с клавиатуры, результат выводится на экран.

Варианты заданий

1. С клавиатуры ввести строку, проверить есть ли в ней буквы A или B.
2. Ввести две строки. Проверить, содержится ли полностью строка 2 в строке 1. Если да, с какой позиции.
3. С клавиатуры ввести строку букв латинского алфавита (длина до 50 символов). Найти количество символов r, w, z (по отдельности) во введенной строке.
4. Вывести на экран первые три слова произвольного текста.
5. С клавиатуры вводится строка S, состоящая из букв и цифр; из букв строки S сформировать строку S1, из цифр – S2.
6. С клавиатуры вводится строка, содержащая одну пару круглых скобок, определить, сколько символов находится внутри скобок.
7. В строке символов выделить первое слово после запятой.
8. Определить в строке количество символов, не являющихся ни буквами, ни цифрами.
9. Определить количество цифр в строке.
10. Для произвольного текста распечатать слова, начинающиеся с буквы 'a'.
11. Вывести первое и последнее слова произвольного текста.
12. В произвольном тексте вставить между первым и вторым словами новое слово.
13. В произвольном тексте вывести слова, содержащие букву 'e'.
14. Определить количество латинских букв в строке.
15. Определить количество чисел с дробной частью в строке.
16. Вывести второе и третье слова произвольного текста.
17. Определить количество сочетаний букв "ти" в произвольном тексте.
18. Определить количество символов в самом длинном слове строки.
19. Проверить имеется ли в заданном тексте баланс открывающих и закрывающих круглых скобок.
20. В произвольном тексте ошибочно сдублированы пробелы. Удалить ненужные символы.

Лабораторная работа № 6. Битовые операции

Цель работы: научиться работать с битами в двоичных кодах.

Работа с двоичными кодовыми комбинациями основывается на использовании сдвигов и логических битовых операциях. Команды сдвигов рассмотрены в описании Лабораторной работы №2. Остановимся на логических операциях.

Команда `NOT` – логическое отрицание

```
NOT приемник
```

Команда `NOT` производит побитовое отрицание операнда приемника.

Операнд приемника может быть регистром или ячейкой памяти.

Пример:

```
mov al,11010000b
```

```
not al ; al=00101111b
```

Команда AND – логическое умножение двух целых чисел

```
AND приемник, источник
```

Команда AND производит побитовое умножение двух операндов и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, признаки SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
and al,10001000b ; al=00001000b
```

Команду удобно использовать для принудительного сброса определенных битов операнда.

```
and bl,11110110b ;сбросить нулевой и третий биты регистра BL

mov eax,00000a5a5h
mov edx,000000ff0h ; маска для установки определённых битов
and eax,edx
```

Команда TEST – сравнение двух целых чисел логическим умножением

```
TEST источник_1, источник_2
```

Команда TEST производит побитовое умножение двух операндов и не сохраняет результат, а устанавливает только биты SF, ZF, и PF в регистре FLAGS в зависимости от результата. Операнд источник_1 может быть регистром или ячейкой памяти. Операнд источник_2 может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
test al,5 ; SF=0;ZF=0;PF=1
```

Эту команду удобно использовать для получения информации о том, являются ли заданные биты операнда нулевыми. Для анализа результата используется флаг ZF, который равен 1, если результат логического умножения равен нулю:

Пример:

```
mov bh,1100b
test bh,0011b ; bh=1100b ZF=1
test bh,1100b ; bh=1100b ZF=0
```

Наиболее часто используется, чтобы узнать равен ли EAX нулю

```
test eax,eax
jz a1
```

Команда OR – логическое сложение двух целых чисел

```
OR приемник, источник
```

Команда OR производит побитовое сложение двух операндов и сохраняет результат в операнде приемника. Операнд приемника может быть регистром или ячейкой памяти. Операнд источника может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к

знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, признаки SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
or al,10001000b ; al=11001101b
```

Команду OR можно использовать для работы с операндами на уровне битов. Типичное использование команды – установка определенных разрядов операнда в единицу.

```
OR BL,00000001b ;установить нулевой бит регистра BL в 1
```

Эту команду можно использовать для сравнения значения регистра с нулевым или ненулевым значением. Например, надо проверить регистр AX, чтобы выяснить равен ли он нулю. Можно было бы применить команду `cmp ax,0`, но используют `or ax,ax` и после этой команды помещают условный переход `jz` или `jnz`. Логическая операция OR, выполненная по отношению к одному и тому же числу, дает в результате это же число. Инструкция OR, как и любая другая математическая инструкция, устанавливает флаги, включая флаг нуля. Но результат выполнения OR будет равен нулю только в том случае, если число в регистре равно нулю.

Команда `cmp eax,0` занимает 3/5 байта и выполняется за 4 такта синхронизации, команда `or eax,eax` занимает 2 байта и выполняется на 1 такт быстрее.

Команда XOR – логическое исключающее ИЛИ двух целых чисел

```
XOR приемник, источник
```

Команда XOR производит побитовое исключающее ИЛИ двух операндов и сохраняет результат в операнде `приемника`. Операнд `приемника` может быть регистром или ячейкой памяти. Операнд `источника` может быть регистром, ячейкой памяти или числом. Операндами команды не могут быть две ячейки памяти. Операнды должны быть одного типа. Если в качестве операнда-источника используется число, то оно автоматически преобразуется к знаковому формату операнда-приемника. Признаки OF, CF устанавливаются в 0, SF, ZF, и PF устанавливаются в соответствии с результатом, признак AF может находиться в любом состоянии.

Примеры:

```
mov al,01001101b
xor al,10001000b ; al=11000101b
```

В программах на языке ассемблера очень часто встречается команда `xor` оба операнда которой представляют собой один и тот же регистр на пример `xor eax,eax`. Это означает обнуление указанного регистра т.е. то же самое что и `mov eax,0`. Команду `xor` для этого используют потому, что она занимает меньше места (2 байта против 5 для команды `mov`) и работает на несколько тактов быстрее.

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Варианты заданий

1. Определить, является ли четным либо нечетным количество единичных разрядов в числе.
2. Вычислить количество единичных разрядов в двоичной записи натурального числа.
3. Определить является введенное с клавиатуры число четным.
4. Проверить, является ли содержимое регистра EAX как последовательность битов палиндромом (т. е. симметричным). Ответ вывести в виде: Да/Нет.

5. Дано X (DWORD) - число в двоичном виде. Необходимо четные разряды задать единицами. Полученное значение разделить на 8 и поменять местами правую и левую части.
6. Проверить, является ли число без знака — содержимое регистра EAX — степенью двойки, т. е. существует k: $0 \leq k \leq 31$, EAX = 2^k. Команды умножения и деления не использовать.
7. Проверить, содержит ли EAX в битовом представлении 6 подряд идущих единиц. Ответ вывести в виде: Да/Нет.
8. Проверить, является ли содержимое регистра EAX числом вида $2^k - 1$. Пример: 00..011…1. Начальная последовательность нулей может отсутствовать. Ответ вывести в виде: Да/Нет
9. Даны X, Y — беззнаковые переменные (WORD), а Z — переменная типа DWORD. Реализовать операцию Z = X + 8*Y, не используя команды умножения.
10. Дано X (DWORD) - число в двоичном виде. Подсчитать количество нулей и количество единиц в двоичном представлении заданного целого неотрицательного числа.
11. Даны X, Y — беззнаковые переменные (WORD), а Z — переменная типа DWORD. Реализовать операцию Z = Y/4 - X, не используя команды деления.
12. Дано X (DWORD) - число в двоичном виде. Необходимо обнулить нечетные разряды числа, а четные заменить на противоположные. Результат разделить на 4 и проинвертировать. Не использовать команду умножения.
13. В регистре EAX заменить 3 левых бита на 3 правых бита, умножить результат на 8 и проинвертировать. Не использовать команду умножения.

Лабораторная работа № 7. Подпрограммы

Цель работы: научиться работать с подпрограммами.

К механизму вызова подпрограммы (в различных языках используется также термины «функция», «метод», «процедура») можно сформировать ряд требований.

1. Возможность передачи управления на произвольный адрес.
2. Возврат управления назад после завершения подпрограммы.
3. Вложенные вызовы подпрограмм.
4. Сохранение и восстановление регистров вызывающей программы.
5. Передача заданного количества аргументов.
6. Возврат значения.
7. Выделение и освобождение памяти под локальные переменные подпрограмм.

В системе команд процессора реализованы только первые три из них. Обеспечить выполнение остальных можно только в том случае, если вызывающая и вызываемая программа «договорятся», где будут находиться передаваемые аргументы и локальные переменные.

Механизм вызова подпрограммы

В системе команд процессора x86 для реализации механизма подпрограмм используются всего две команды:

- команда вызова подпрограммы `call`, единственным аргументом которой является адрес начала подпрограммы;

- команда возврата из подпрограммы `ret`.

Пусть следующая команда, расположенная по адресу — `call f`. Где `f` адрес возврата, после чего в указатель команд `ip` помещается адрес, так что эта команда становится следующей для исполнения процессором. Когда в процессе исполнения подпрограммы `f` встретится команда `ret`, из стека извлекается верхнее машинное слово — там должен быть адрес возврата — и помещается в указатель команд `ip`. Соответственно, выполнение вызывающей программы продолжится со следующей за `call` командой.

Большинство существующих процессоров поддерживают работу со стеком на уровне машинных команд. Команды работы со стеком позволяют заносить в стек и извлекать из него двухбайтные "слова" и четырёхбайтные "двойные слова" (отдельные байты записывать в стек нельзя, так что адрес вершины стека всегда остаётся чётным).

Регистр `ESP`, формально относящийся к группе регистров общего назначения, тем не менее практически никогда не используется ни в какой иной роли, кроме роли указателя стека. Название этого регистра как раз и означает "stack pointer".

Считается, что адрес, содержащийся в `ESP`, указывает на вершину стека, то есть на ту область памяти, где хранится последнее занесённое в стек значение. Стек "растёт" в сторону уменьшения адресов, то есть при занесении в стек нового значения `ESP` уменьшается, при извлечении значения – увеличивается.

Занесение значения в стек производится командой `push`, имеющей один операнд. Этот операнд может быть непосредственным, регистровым или типа "память" и иметь размер `word` или `dword` (если операнд не регистровый, то размер необходимо указать явно). Для извлечения значения из стека используется команда `pop`, операнд которой может быть регистровым или типа "память". Естественно, операнд должен иметь размер "слово" или "двойное слово".

Команды `push` и `pop` совмещают копирование данных (на вершину стека или с неё) со сдвигом самой вершины, то есть изменением значения регистра `ESP`. Понятно, что можно, вообще говоря, обратиться к значению на вершине стека, не извлекая его из стека – применив (в любой команде, допускающей операнд типа "память") операнд `[esp]`.

Например, команда

```
mov eax,[esp]
```

скопирует четырёхбайтное значение с вершины стека в регистр `EAX`.

Как говорилось выше, стек очень удобно использовать для временного хранения значений из регистров:

```
push eax ; запоминаем eax
; . . . используем eax под посторонние нужды . . .
pop eax ; восстанавливаем eax
```

Рассмотрим более сложный пример. Пусть регистр `ESI` содержит адрес некоторой строки символов в памяти, причём известно, что строка заканчивается байтом со значением 0 (но неизвестно, какова длина строки) и нам необходимо "обратить" эту строку, то есть записать составляющие её символы в обратном порядке в том же месте памяти. Нулевой байт, играющий роль ограничителя, естественно, остаётся при этом на месте и никуда не копируется. Один из способов сделать это – последовательно записать коды символов в стек, а затем снова пройти строку с начала в конец, извлекая из стека символы и записывая их в ячейки, составляющие строку.

Поскольку записывать в стек однобайтовые значения нельзя, мы будем записывать значения двухбайтовые, причём старший байт просто не будем использовать. Конечно, можно сделать всё более рационально, но нам в данном случае важнее наглядность нашей иллюстрации. Для промежуточного хранения будем использовать регистр `BX`, причём только его младший байт (`BL`) будет содержать полезную информацию, но записывать в

стек и извлекать из стека мы будем весь BX целиком. Задача будет решена в два цикла. Перед первым циклом мы занесём ноль в регистр ECX, потом на каждом шаге будем извлекать байт по адресу [esi+ecx] и помещать этот байт (в составе слова) в стек, а ECX увеличивать на единицу, и так до тех пор, пока очередной извлечённый байт не окажется нулевым, что по условиям задачи означает конец строки. В итоге все ненулевые элементы строки окажутся в стеке, а в регистре ECX будет длина строки.

Поскольку для второго цикла заранее известно количество его итераций (длина строки) и оно уже содержится в ECX, мы организуем этот цикл с помощью команды loop. Перед входом в цикл мы проверим, не пуста ли строка (то есть не равен ли ECX нулю), и, если строка была пуста, сразу же перейдём в конец нашего фрагмента. Поскольку значение в ECX будет уменьшаться, а строку нам нужно пройти в прямом направлении " наряду с ECX мы воспользуемся регистром EDI, который в начале установим равным ESI (то есть указывающим на начало строки), а на каждой итерации будем его сдвигать. Итак, пишем:

```
     xor ebx,ebx ; обнуляем ebx
     xor ecx,ecx ; обнуляем ecx
     lp : mov bl , [esi+ecx] ; очередной байт из строки
     cmp bl , 0 ; конец строки?
     je lpquit ; если да - конец цикла
     push bx ; bl нельзя , приходится bx
     inc ecx ; следующий индекс
     jmp lp ; повторить цикл
lpquit: jecxz done ; если строка пустая - конец
     mov edi,esi ; опять с начала буфера
lp2: pop bx    ; извлекаем
     mov [edi],bl ; записываем
     inc edi   ; следующий адрес
     loop lp2   ; повторять ecx раз
done:
```

Директива PROC. Определение процедуры

Процедуру можно определить как именованный блок команд, оканчивающийся оператором возврата. Для объявления процедуры используются директивы PROC и ENDP. При объявлении процедуре должно быть назначено имя, которое является одним из разрешенных идентификаторов. В каждой из рассмотренных ранее программ была только одна главная процедура под названием main, например:

```
main PROC
```

При создании любых других процедур, не являющихся стартовыми, нужно в их конце разместить команду RET. В результате процессор вернет управления команде, следующей за той, которая вызвала эту процедуру:

```
sample PROC
ret
sample ENDP
```

К особому типу процедур относится так называемая стартовая процедура программы, которой назначено имя main, поскольку она должна завершаться не командой RET, а вызовом функции ExitProcess системы Windows, которая и завершает выполнение программы:

```
INVOKE ExitProcess, 0
```

В отличие от команды процессора CALL, директива INVOKE является встроенным оператором ассемблера, позволяющим вызывать указанную процедуру и передавать ей параметры.

Пример процедуры: суммирование трех целых чисел

Создадим процедуру SumOf, вычисляющую сумму трех 32-разрядных чисел. Предположим, что перед вызовом процедуры значения этих чисел мы должны поместить в регистры EAX, EBX и ECX, а сумма возвращается в регистре EAX:

```
SumOf PROC
add eax, ebx
add eax,ecx
ret
 SumOf ENDP
```

Ниже приведены несколько рекомендаций по оформлению текстов процедур. В начале каждой процедуры следует помесить:

• Описание всех функций, выполняемых процедурой.

• Список входных параметров и описание их значений. Если какой-либо из параметров имеет особый тип, его нужно также указать. Обычно входные параметры указываются после ключевого слова Передается (Receives).

• Список возвращаемых процедурой значений, указанных после ключевого слова Возвращается (Returns).

• Перечень особых требований (если таковые имеются), которые должны быть удовлетворены перед вызовом процедуры (входные условия).

ПРИМЕР оформления текста процедуры.

```
SumOf PROC
;Вычисляет и возвращает сумму трех 32-разрядных целых чисел.
;Передается: три числа в регистрах EAX, EBX, ECX.
;Возвращается: сумма в регистре EAX, а также флаги состояния
   ;(переноса, переполнения и др.)
add eax, ebx
add eax, ecx
ret
SumOf ENDP
```

Команды CALL и RET

Команда CALL предназначена для передачи управления процедуре, адрес которой указывается в качестве параметра. При этом процессор начинает выполнять команду, расположенную по указанному адресу. Чтобы вернуть управление команде, расположенной сразу за CALL, в процедуре используется команда RET.

Создадим процедуру под именем ArraySum, которой из вызывающей программы будут передаваться два параметра: указатель на массив 32-разрядпых целых чисел и количество элементов в этом массиве. Сумму элементов массива процедура будет возвращать в регистре EAX:

```
ArraySum PROC
;Вычисляет сумму элементов массива 32-разрядным целых чисел
;Передается: ESI = адрес массива
; ECX = количество элементов массива
; Возвращается: EAX = сумма элементов массива
push esi ;Сохраним значения регистров ESI и ECX
push ecx
mov eax,0 ;Обнулим значение суммы
L1 :
add eax,[esi] ;Прибавим очередной элемент массива
add esi,4 ;Вычислим адрес следующего элемента массива
loop L1  ;Повторим цикл для всех элементов массива
pop ecx  ;Восстановим значения регистров ESI и ECX
```

```
pop esi
ret    ;Вернем сумму в регистре EAX
ArraySum ENDP
```

Передача параметров по значению

Процедуре передается собственно значение параметра. При этом фактически значение параметра копируется, и процедура использует его копию, так что модификация исходного параметра оказывается не возможной. Этот механизм применяется для передачи небольших параметров, таких как байты или слова, к примеру, если параметры передаются в регистрах:

```
mov ax,word ptr value ; Сделать копию значения,
call procedure    ; Вызвать процедуру.
```

Передача параметров по ссылке

Процедуре передается не значение переменной, а ее адрес, по которому процедура сама прочитает значение параметра. Этот механизм удобен для передачи больших массивов данных и в тех случаях, когда процедура должна модифицировать параметры (хотя он и медленнее из-за того, что процедура будет выполнять дополнительные действия для получения значений параметров).

```
mov ax,offset value
call procedure
```

Передача параметров по возвращаемому значению

Этот механизм объединяет передачу по значению и по ссылке. Процедуре передают адрес переменной, а процедура делает локальную копию параметра, затем работает с ней, а в конце записывает локальную копию обратно по переданному адресу. Этот метод эффективнее обычной передачи параметров по ссылке в тех случаях, когда процедура должна обращаться к параметру очень большое количество раз, например, если используется передача параметров в глобальной переменной:

```
mov . global_variable,offset value
call procedure
[...]
procedure proc near
mov dx,global_variable
mov ax, word ptr [dx] ;команды, работающие с AX в цикле
          ;десятки тысяч раз
mov word ptr [dx],ax
procedure endp
```

Передача параметров по результату

Этот механизм отличается от предыдущего только тем, что при вызове процедуры предыдущее значение параметра никак не определяется, а переданный адрес используется только для записи в него результата.

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 применяют 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 активизационной 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 активизационной записи (то есть от `BP`), например так:

```
push X
push Y
push Z
call xyzzy
xyzzy proc
xyzzy_z equ
xyzzy_y equ
xyzzy_x equ
push
mov
near
[bp+8]
tbp+6]
[bp+4]
bp
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 последовательными командами `CALL` вызвать несколько функций с одними и теми же параметрами.

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Подготовить программу в соответствии с вариантом лабораторного задания к работе № 5. Программа должна быть подготовлена для выполнения в среде masm32.

Лабораторная работа № 8. Макросы

Цель работы: научиться работать с макрокомандами.

Макропроцедурой (macro procedure) называется именованный блок команд языка ассемблера. После того как макропроцедура определена в программе, ее можно многократно вызывать в разных участках кода. При вызове макропроцедуры, в код программы будут помещены содержащиеся в ней команды. Не следует путать вызов макропроцедуры с вызовом обычной процедуры, поскольку в первом случае команда `CALL` не используется.

Следует отметить, что термин макропроцедура используется в документации к компилятору Microsoft Assembler для обозначения макрокоманд, не возвращающих значения. Кроме макропроцедур, существуют также макрофункции (macro functions), возвращающие значение. В среде программистов прижился термин макрокоманда, который, по сути, эквивалентен термину макропроцедура. Поэтому далее будет использоваться термин макрокоманда.

Макроопределение

Размещение. Определения макрокоманд, или макроопределения, помещаются либо непосредственно в текст исходной программы на ассемблере (как правило, в его начало), либо в отдельный текстовый файл, который включается в исходную программу на этапе компиляции с помощью директивы `INCLUDE`.

Как только в тексте программы встречается имя макрокоманды, оно заменяется препроцессором на соответствующий набор команд, который указан в ее макроопределении. В приведенном ниже примере макрокоманда `NewLine` генерирует одну ассемблерную команду, которая вызывает библиотечную процедуру `CrLf`:

```
NewLine MACRO
call CrLf
 ENDM
```

Текст этого определения обычно помещается непосредственно перед сегментом данных. После этого в сегменте кода макрокоманда вызывается так:

```
.code
NewLine
```

При обработке программы препроцессором вызов макрокоманды `NewLine` будет заменен приведенной ниже командой:

```
call CrLf
```

В данном случае произошла обычная текстовая подстановка, которую можно было бы выполнить и с помощью директивы `TEXTEQU`. У макрокоманды, так же, как и у процедуры, может быть один или несколько параметров, что делает ее гораздо более мощным средством по сравнению с директивой `TEXTEQU`.

Определение макрокоманды

Макрокоманду можно определить в любом месте исходного кода программы, воспользовавшись директивами `MACRO` и `ENDM`. Синтаксис макроопределения, следующий:

```
Имя MACRO Параметр-1, Параметр-2... Список-команд
ENDM
```

Рекомендуется всегда выделять отступами те команды, которые помещаются между директивами `MACRO` и `ENDM`, чтобы подчеркнуть их принадлежность к макроопределению. То же самое касается и выбора имен макрокоманд. Для их выделения рекомендуется пользоваться специальным префиксом. Далее для выделения имен макрокоманд перед

ними помещается префикс в виде строчной буквы "m", например, mPutchar, mWriteString и mGotoXY.

Команды, находящиеся между директивами MACRO и ENDM, до вызова макрокоманды не компилируются. Макроопределение может содержать произвольное количество параметров, которые разделяются запятыми.

Пример макроопределения mPutchar. Рассмотрим макрокоманду mPutchar, имеющую один входной параметр, имя которого char. Данная макрокоманда выводит символ, переданный ей в качестве параметра, на терминал с помощью вызова процедуры WriteChar:

```
mPutchar MACRO char
push eax
mov al,char
call WriteChar
pop eax
 ENDM
```

Вызов макрокоманд

Для вызова макрокоманды нужно поместить ее имя в исходный код программы и при необходимости указать передаваемые ей значения:

```
Имя_макрокоманды Значение-1, Значение-2, ...
```

Имя_макрокоманды должно быть определено в исходном коде программы до ее вызова. Каждое значение является обычной текстовой строкой, которое подставляется вместо соответствующего параметра макрокоманды. Порядок передачи значений параметров должен соответствовать порядку их следования в макроопределении, однако число значений необязательно должно соответствовать числу параметров макрокоманды.

Часто повторяющуюся в программе последовательность команд можно описать как макрокоманду транслятора.

Описание макрокоманды

```
Имя макрокоманды MACRO [fl,J2,...] ;директива начала описания команды
ENDM ; директива конца описания
```

fl, J2 - входные символические параметры (необязательные) Вызов макрокоманды: Имя макрокоманды [t1, t2,…]

t1,t2 - фактические параметры (если были входные)

Когда транслятор встречает в кодовом сегменте вызов макрокоманды, он просто подставляет в этом месте последовательность команд из описания макрокоманды.

Макросредства транслятора позволяют сократить исходный текст программы, но не машинный код - в отличие от процедуры.

Правила использования макрокоманд

■ Описание макрокоманды делается в начале исходного текста до описания сегментов

■ Если в теле макрокоманды есть метки (то есть, разветвления), в описании макрокоманды должна присутствовать директива LOCAL с перечислением используемых меток. Это позволит избежать дублирования меток, когда макрокоманда вызывается несколько раз.

Пример В фрагменте программы надо обнулить две различные по длине области памяти в сегменте данных.

Создадим макрокоманду NUL, выполняющую обнуление произвольного массива в памяти. Используем формальные параметры в макрокоманде: адрес массив - adress и длина массива в байтах - count

```
      ; ====описание макрокоманды NUL=====:=
      nul macro adress, count
      local m1
      lea si,adres
      mov cx,count
m1:   mov byte ptr ds:[si],0
      inc si
      loop m1
      endm
      ; в сегменте данных
      obl1 db 16 dup(?)
      obl2 db 8 dup(?)
      ; в кодовом сегменте
      nul obl1,16 ; вызов макрокоманды
      nul obl2,8
```

Лабораторные задания

Все переменные вводятся с клавиатуры, результат выводится на экран. Переменные имеют формат слова.

Варианты заданий

Подготовить программу в соответствии с вариантом лабораторного задания к работе № 6. Программа должна быть подготовлена для выполнения в среде masm32.