Практическая работа 3 Арифметические операции над разными типами данных

Цель научиться размещать массивы данных в памяти и выполнять арифметические действия над разными типами данных.

Необходимая Теория

Сегменты и директивы сегментации

Программа на языке Assembler в соответствии с особенностями архитектуры компьютера (микропроцессора) состоит из сегментов.

Вспомним, что физически сегмент представляет собой область (блок) памяти, занятую командами и/или данными.
Адреса сегментов (адрес начала сегмента) хранятся в соответствующих сегментных регистрах.
Адреса команд/данных вычисляются относительно начала сегмента.

Микропроцессор имеет шесть сегментных регистров, посредством которых может одновременно работать:

- с одним сегментом кода (CS);
- с одним сегментом стека (SS);
- с одним сегментом данных (DS);
- с тремя дополнительными сегментами данных (ES,FS, GS).

Итак, исходный текст программы на языке Assembler разбивается на сегменты. Каждая программа содержит как минимум сегмент данных, сегмент стека, сегмент кода.

Для описания сегментов предназначены ***директивы сегментации***.
Директивы сегментации подразделяются на

1) ***стандартные*** (поддерживаются всеми трансляторами Assembler);
2) ***упрощенные*** (поддерживаются транслятором TASM).

Стандартные директивы сегментации
Стандартно сегменты на языке Assembler описываются с помощью директивы SEGMENT.

Синтаксическое описание сегмента представляет собой следующую конструкцию

<имя сегмента> SEGMENT [тип выравнивания] [тип комбинирования]
[класс сегмента] [тип размера сегмента]
<тело сегмента>
<имя сегмента> ENDS

Параметры имеют множество значений, которыми при желании можно ознакомиться в справочниках команд.

С помощью директивы ASSUME можно сообщить транслятору, какой сегмент, к какому сегментному регистру «привязан», или, говоря боле точно, в каком сегментном регистре хранится адрес сегмента..

Формат директивы ASSUME:

ASSUME <сегментный регистр>:<имя сегмента>

Директивы **SEGMENT и ASSUME - стандартные директивы сегментации**.

Для простых программ, которые содержат по одному сегменту кода, данных и стека легче использовать упрощенные описания соответствующих сегментов.

Трансляторы MASM и TASM предоставляют возможность использования упрощенных директив сегментации (вместо SEGMENT).

Замечание. Стандартные и упрощенные директивы сегментации не исключают друг друга.

Совет 1. Стандартные директивы используются, когда программист хочет получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Совет 2. Упрощенные директивы целесообразно использовать

1) для простых программ
2) программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня (это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления).

Упрощенные директивы определения сегмента
(для режима MASM)

.CODE [имя]
Директива предназначена для определения начала или продолжения сегмента кода. Возможно определение нескольких сегментов данного типа.

.DATA
Директива предназначена для определения начала или продолжения сегмента инициализированных данных. Также используется для определения данных типа near.

.STACK [размер]
Директива предназначена для определения начала или продолжения сегмента стека. Параметр [размер] задает размер стека.

.CONST
Директива предназначена для определения начала или продолжения сегмента постоянных данных (констант).

.DATA?
Директива предназначена для определения начала или продолжения сегмента неинициализированных данных. Также используется для определения данных типа near

.FARDATA [имя]
Директива предназначена для определения начала или продолжения сегмента инициализированных данных типа far. Возможно определение нескольких сегментов данного типа.

.FARDATA? [имя]
Директива предназначена для определения начала или продолжения сегмента неинициализированных данных типа far. Возможно определение нескольких сегментов данного типа.

Для режима IDEAL директивы носят следующие названия (в порядке упоминания)
CODESEG[имя] DATASEG STACK [размер] CONST UDATASEG
FARDATA [имя] UFARDATA [имя]

Совместно с упрощенными директивами сегментации используется **директива указания модели памяти MODEL**, которая частично управляет размещением сегментов и выполняет функции директивы ASSUME, т.е. связывает сегменты с сегментными регистрами (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать).
Замечание. Необходимо явно инициализировать только регистр ds.

Упрощенный формат директивы MODEL

MODEL [<модификатор>] <модель памяти> [др. параметры]

Обязательным параметром директивы MODEL является «модель памяти». Этот параметр определяет модель сегментации памяти для программного модуля.

Возможные значения параметра «модель памяти»:

TINY - Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата *com*.

SMALL - Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на языке Assembler.

MEDIUM - Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления типа far. Данные объединены в одной группе, все ссылки на них типа near.

COMPACT - Код в одном сегменте; ссылка на данные типа far.

LARGE - Код в нескольких сегментах. Каждый объединяемый в одну программу модуль хранится в отдельном сегменте кода.

Параметр «модификатор» директивы MODEL уточняет особенности использования выбранной модели памяти.

Возможные значения параметра «модификатор модели памяти»:
use16 - сегменты выбранной модели используются как 16-битные
use32 - сегменты выбранной модели используются как 32-битные
dos - программа предназначена для работы в ОС MS-DOS

Другие параметры используются при написании программ на разных языках программирования (пока использовать не будем).

Пример.
Для большинства программ на ассемблере используют директиву

MODEL SMALL

При использовании директивы MODEL транслятор создает и делает доступными для программиста идентификаторы, в которых хранится информация о физических адресах сегментов.
Перечислим идентификаторы, создаваемые директивой MODEL:
@code - физический адрес сегмента кода
@data - физический адрес сегмента данных типа near
@fardata - физический адрес сегмента данных типа far
@fardata? - физический адрес сегмента неинициализированных данных типа far
@curseg - физический адрес сегмента неинициализированных данных типа far
@stack - физический адрес сегмента стека

Структура программы на языке Assembler

Таким образом, общая структура программы может выглядеть следующим образом:

```
        masm                ;режим работы TASM: ideal или masm
        model  small        ;модель памяти
        .stack <размер>     ;сегмент стека
        .data               ;сегмент данных
               <описание данных>
BEGIN : .code               ;сегмент кода
               <команды>
        end  BEGIN          ;конец программы с точкой входа BEGIN
```

Пример.
Текст программы с использованием упрощенных директив сегментации.

Описаны 3 сегмента программы: сегмент данных, сегмент стека и сегмент кода.
В сегменте данных задана строка для вывода на экран.
Размер стека равен 256 байт.
Сегмент кода содержит команды начала и завершения программы, а также комментарии к пропущенным командам.

```
masm                    ;режим работы TASM: ideal или masm
model  small            ;модель памяти
.data                   ;сегмент данных
message db  'Привет всем,$'
.stack                  ;сегмент стека
        db    256   dup ('?')     ;сегмент стека
.code                   ;сегмент кода
main :          ;начало программы
      mov   ax,@data      ;заносим в сегментный регистр ds
      mov   ds,ax         ; физический адрес сегмента данных
          ;здесь будут команды вывода строки на экран
      …….
        ; завершение программы
      mov   ax,4c00h      ;пересылка 4c00h в регистр ax
      int   21h           ;вызов прерывания с номером 21h
end   main                ;конец программы с точкой входа main
```

В модели SMALL описание сегмента данных производится с помощью упрощенной директивы сегментации . DATA.

 ператорOFFSET возвращает адрес переменной. Он используется для указания местоположения, а не содержимого переменной:

```
.DATA
   MyVar DB 77h; переменная байтового размера MyVar, инициализированная значением 77h
.CODE
   .
   mov eax, MyVar; скопировать 77h в eax
   mov ebx, смещение MyVar; скопировать адрес памяти, где значение 77h сохранено в ebx
```

 OFFSET также может передавать адрес переменной в *процедуру* в операторе INVOKE .
 ОднакоOFFSET будет работать только для *глобальных переменных,* объявленных в .DATA или .DATA? сегменты.
 OFFSET завершится ошибкой с *локальными переменными* , которые объявляются при входе в процедуру с помощью оператора LOCAL .
 Локальные переменные внутри процедуры не имеют смещения, потому что они создаются*в стеке во время выполнения* .
 ОператорADDR решает эту проблему. Он используется исключительно с INVOKE для передачи *адреса переменной* в процедуру.
 Для*глобальных переменных* оператор ADDR преобразуется в простую инструкцию PUSH , как если бы использовалось OFFSET :

```
нажать GlobalVar
```

 Однако для*локальных переменных* ADDR преобразуется в: lea eax, LocalVar; загрузить эффективный адрес LocalVar в eax push eax
 Фактический адрес- это физический адрес данных в памяти.
 Важно помнить, что при использовании ADDR с локальными переменньми регистр EAX модифицируется, а не остается доступным для других целей в вызывающей процедуре.

Некоторые операции

Загрузка в регистры (reg) и ячейки памяти (mem), возможна косвенная адресация [mem] и косвенная с смещением [mem+index], допускаются варианты составного и комбинированного индексов

```
mov reg,reg
mov reg,mem
mov reg,[mem]
mov mem,reg
mov [mem],reg
mov reg,[mem+index]
mov reg,[reg+index]
```

Сложение

```
add reg,reg
add reg,mem
add reg,[mem]
```

Вычитание

```
sub reg,reg
sub reg,mem
sub reg,[mem]
```

inc reg – инкремент
dec reg – декремент

переходы
jmp address ; Безусловный переход

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

Сравнение
cmp reg,mem
cmp reg,reg

Пример
cmp AX,10000 ;AX-10000
je eql0000 ;Переход на метку eql0000, если AX=10000

Сдвиги

Логический сдвиг операнда влево/вправо

```
SHL операнд, количество_сдвигов
SHR операнд, количество_сдвигов
```

SHL и SHR сдвигают биты операнда (регистр/память) влево или вправо соответственно на один разряд и изменяют флаг переноса cf. При логическом сдвиге **все биты равноправны, а освободившиеся биты заполняются нулями.** Указанное действие повторяется количество раз, равное значению второго операнда.

Арифметический сдвиг влево/вправо

```
SAL операнд, количество_сдвигов
SAR операнд, количество_сдвигов
```

Команда SAR — сдвигает биты операнда (регистр/память) вправо на один разряд, значение последнего вытолкнутого бита попадает в флаг переноса, а освободившиеся биты заполняются знаковым битом.

Команда SAL — сдвигает биты операнда (регистр/память) влево на один разряд, значение последнего вытолкнутого бита попадает в флаг переноса, а освободившиеся биты заполняются нулями, при этом знаковый бит не двигается**.**

Команды циклического сдвига

```
rol операнд, количество_сдвигов
ror операнд, количество_сдвигов
rcl операнд, количество_сдвигов
rcr операнд, количество_сдвигов
```

ROL и ROR сдвигают все биты операнда влево(для ROL) или вправо(для ROR) на один разряд, при этом старший(для ROL) или младший(для ROR) бит операнда вдвигается в операнд справа(для ROL) или слева(для ROR) и становится значением младшего(для ROL) или старшего(для ROR) бита операнда; одновременно выдвигаемый бит становится значением флага переноса cf**. Указанные действия повторяются количество раз, равное значению второго операнда.**

RCL и RCR сдвигают все биты операнда влево (для RCL) или вправо (для RCR) на один разряд, при этом старший(для RCL) или младший(для RCR) бит становится значением флага переноса cf; одновременно старое значение флага переноса cf вдвигается в операнд справа(для RCL) или слева(для RCR) и становится значением младшего(для RCL) или старшего(для RCR) бита операнда. **Указанные действия повторяются количество раз, равное значению второго операнда.**

Задание

Ниже приведен код выполняющий чтение данных из файла и некоторые аспекты синтаксиса MASM. Разберитесь в коде прграммы

```
.686P
.MODEL FLAT, STDCALL
.STACK 4096  ;размер стека

option casemap:none
include c:\masm32\include\windows.inc  ;обратите внимание на путь к файлам inc и lib
include c:\masm32\include\user32.inc
includelib c:\masm32\lib\user32.lib
include c:\masm32\include\kernel32.inc
includelib c:\masm32\lib\kernel32.lib

.DATA ;сегмент инициализированных данных
FileName DB "w_512.dat",0  ;здесь лучше указывать полный путь к файлу
BadText db "File dont open!",0

.DATA?  ;сегмент не инициализированных данных
hFile    HANDLE ?
hMemory  DWORD  ?
pMemory  DWORD  ?
memID  DWORD  ?
SizeR    DWORD  ?
dwBytesRead   dd    ?
```

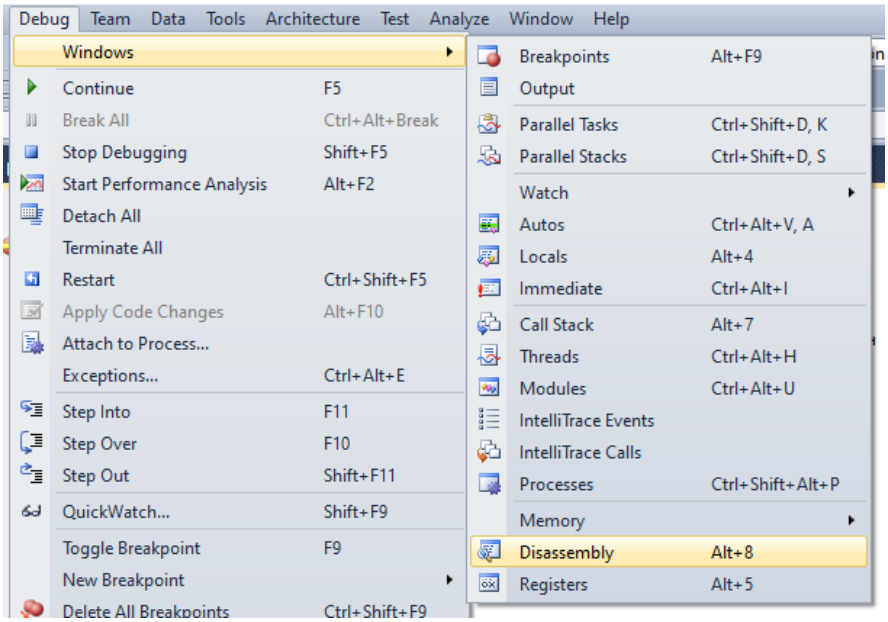
```
HW      DD ?
dwFileSize dd ?

;EXTERN MessageBoxA@16:NEAR ;так комментируется одна строка строка

.CODE ;сегмент кода
START: ;точка старта программы

; открываем файл для чтения через API функцию
invoke CreateFile, addr FileName, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL

mov hFile, eax  ;возвращаем хендел файла
cmp hFile, INVALID_HANDLE_VALUE ;проверяем на хендел файла на валидность
jz ErrorMsg

invoke GetFileSize, hFile, NULL ;определяем размер файла
mov    dwFileSize, eax

INVOKE GlobalAlloc, GMEM_FIXED or GMEM_ZEROINIT, dwFileSize
mov  hMemory, eax
invoke GlobalLock, hMemory
mov pMemory, eax ; Возвращаем указатель на память


;читаем данные из файла размером 2048 байт
INVOKE ReadFile, hFile, pMemory, 2048, addr dwBytesRead, NULL ;размер чтения должен быть не болше размера
стека
; колво прочитанных байт будет возвращено в dwBytesRead
or     eax, eax ; проверяем на ошибку чтения
jz     ErrorMsg

mov bx, word ptr [pMemory];так не работает ПОЧЕМУ????

mov eax, pMemory;Грузим указатель
mov bx, [eax];так читаем первое слово
mov bx, [eax+2];так читаем второе слово
mov ebx, [eax];так читаем как integer с первого байта буфера памяти
mov ebx, [eax+1];так читаем как integer со второго байта буфера памяти
mov bx,  [eax+512*2];так читаем 512 слово
mov bl,  [eax];так читаем первый байт
mov bh,  [eax+1];так читаем второй байт

push eax
add eax,2048
mov memID,eax
pop eax

INVOKE ReadFile, hFile, memID, 2048, addr dwBytesRead, NULL ;размер чтения должен быть не больше размера
стека
; колво прочитанных байт будет возвращено в dwBytesRead
or     eax, eax ; проверяем на ошибку чтения
jz     ErrorMsg

;mov bx, [eax+2048];ошибка чтения
; или так
mov eax,memID
mov cx, [eax];первый байт


invoke GlobalUnlock, pMemory
invoke GlobalFree, hMemory
INVOKE CloseHandle, hFile


jmp End_code
```

```
COMMENT * ; так оформляется многострочный комментарий
 PUSH     OFFSET STR1
 PUSH     OFFSET STR2
 PUSH     HW
 CALL     MessageBoxA@16
 *
 ErrorMsg:
   invoke MessageBox,NULL,addr BadText,addr BadText,MB_OK
   invoke ExitProcess,0
 End_code:
RET
END START
```

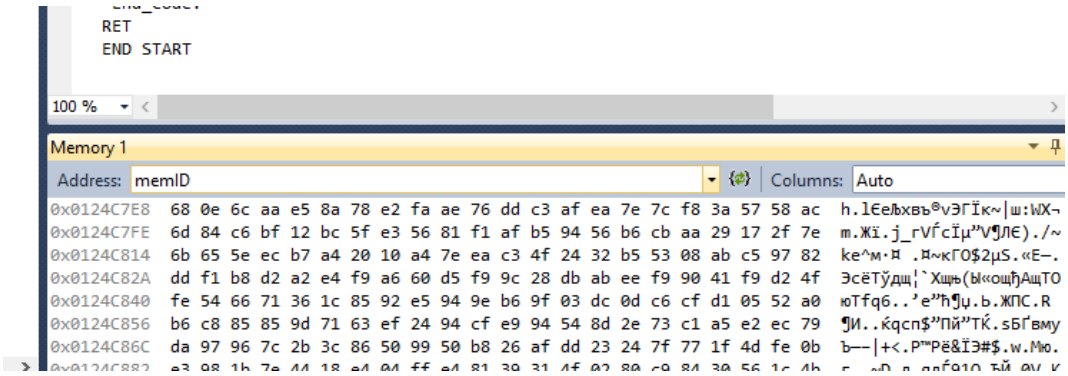
Модифицируйте программу для выполнения следующих заданий.

1. Рассчитайте контрольную сумму для первых 256 слов (word) прочитанных из файла w_512.dat. Контрольная сумма рассчитывается по следующим образом:
   - суммируются все 256 слов. Результат берется по модулю 0x10000h, затем из 0 вычитается полученный результат. Полученное после вычитания значения будем считать контрольной суммой (CRC16).
2. Рассчитайте контрольную сумму для первых 256 двойных слов (dword) прочитанных из файла w_512.dat. В этом случае сумма берется по модулю 0x100000000h, затем из 0 вычитается полученный результат. Полученное после вычитания значения будем считать контрольной суммой (CRC32).
3. Для 512 значений типа word, четные слова с нечетными. Результат сохраните в отдельный массив в памяти.
4. Создайте массив в который сохраните все нечетные значения из первых 1024 элементов типа dword прочитанных из файла. Выведите количество полученных элементов.
5. Выведите последние два байта прочитанные из файла.
6. Выведите первый байт сдвинутый на бит влево и первое слово сдвинутое на 3 бита вправо (сдвигать инструкциями циклического сдвига, логического сдвига и арифметического сдвига).

Контроль данных произвести в отладчике.
Когда производится пошаговая отладка программисту доступны следующие окна: память, дизассемблер и регистры.



Загружаемые даны можно проконтролировать в окне память, для этого необходимо указать адрес или название переменной, которая ссылается на участок памяти



В отчете – краткий листинг программы и значения, полученные в соответствии с пунктом задания. Выполнить все 6 заданий.