Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

В. А. Савельев, И. В. Дорощенко

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Рекомендовано учебно-методическим объединением по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для студентов учреждений высшего образования по специальности 6-05-0611-01 «Информационные системы и технологии»

Электронный аналог печатного издания

Гомель 2025

УДК 004.4'424(075.8)
ББК 32.973.2я73
С13

Рецензенты: зав. кафедрой автоматизированных систем обработки информации ГГУ им. Ф. Скорины канд. техн. наук, доц. *А. В. Воруев*;
зам. гл. конструктора ОАО «СтанкоГомель» *Ю. Л. Аникейчик*

Савельев, В. А.

С13 Программирование на языке Ассемблера : учебно-методическое пособие / В. А. Савельев, И. В. Дорощенко ; Министерство образования Республики Беларусь, Гомельский государственный технический университет имени П. О. Сухого. – Гомель : ГГТУ им. П. О. Сухого, 2025. – 122 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

ISBN 978-985-535-557-2.

Учебно-методическое пособие содержит примеры решения типовых задач программирования микропроцессоров по дисциплине «Программирование на языке Ассемблера», а также задания для лабораторных работ.

Для студентов специальности 6-05-0611-01 «Информационные системы и технологии».

УДК 004.4'424(075.8)
ББК 32.973.2я73

ISBN 978-985-535-557-2

Оглавление

Глава 1. Структура программы на языке Ассемблера

Далее будем полагать, что имеем дело с 32-битными процессорами, начиная с *Intel* 80386. Эти процессоры имеют 32-битную шину адреса, следовательно, могут адресовать 2^{32} = 4 294 967 296 = 4 Гб памяти.

Каждую Win32 программу Windows запускает в отдельном виртуальном пространстве. Это означает, что каждая Win32 программа будет иметь 4-гигабайтовое адресное пространство, но вовсе не означает, что каждая программа имеет 4 Гб физической памяти, а только то, что программа может обращаться по любому адресу в этих пределах.

Под Win32 есть только одна модель память: плоская, без 64-килобайтных сегментов. Память – это большое последовательное 4-гигабайтовое пространство. Можно не использовать сегментные регистры, зато можно использовать любой сегментный регистр для адресации к той или иной точке памяти.

При программировании под Win32 необходимо помнить, что Windows использует регистры *esi*, *edi*, *ebp* и *ebx* для своих целей и не ожидает, что вы измените их значение. Если же вы используете какой-либо из этих четырех регистров в вызываемой функции, то не забудьте восстановить их перед возвращением управления Windows.

Операционная система Windows предоставляет огромное количество ресурсов Windows-программам через Windows API (Application Programming Interface) – коллекцию полезных функций, располагающихся непосредственно в операционной системе и готовых для использования программами. Эти функции находятся в нескольких динамически подгружаемых библиотеках (DLL), таких как *kernel32.dll*, *user32.dll* и *gdi32.dll*. *Kernel32.dll* содержит API функции, взаимодействующие с памятью и управляющие процессами, *user32.dll* контролирует пользовательский интерфейс, *gdi32.dll* отвечает за графические операции. Существуют также другие библиотеки. Код API-функций не включается в исполняемый файл.

Вызов системных функций API Win32 из программы на ассемблере подчиняется набору соглашений *stdcall*. Эти соглашения заключаются в следующем:

– **аргументы передаются вызываемой функции через стек**. Если аргумент укладывается в 32-битное значение и не подлежит модификации вызываемой функцией, он обычно записывается в стек непосредственно. В остальных случаях программист должен размес-

тить значение аргумента в памяти, а в стек записать 32-битный указатель на него. Таким образом, все передаваемые функции API параметры представляются 32-битными величинами, и количество байт, занимаемых в стеке для передачи аргументов, кратно четырем;

– в соглашении о вызовах *stdcall* порядок передачи параметров осуществляется справа налево, т. е. первый параметр помещается в стек последним. Функция, вызывающаяся с использованием этого соглашения, должна очистить стек после вызова. После загрузки всех параметров **программа вызывает функцию командой *call***;

– ответственность за сохранение и восстановление регистров лежит на вызывающей функции, а не на вызываемой функции. То есть вызывающая функция должна сохранить значения регистров, которые ей необходимы, перед вызовом функции и восстановить их после возврата из вызываемой функции;

– вызываемая функция API гарантированно сохраняет регистры общего назначения *ebp*, *esi*, *edi*. Регистр *eax*, как правило, содержит возвращаемое значение. Состояние остальных регистров после возврата из функции API следует считать неопределенным. (Полный набор соглашений *stdcall* регламентирует также сохранение системных регистров *ds* и *ss*. Однако для *flat*-модели памяти, используемой в Win32, эти регистры значения не имеют.)

Итак, вот шаблон программы на Ассемблере MASM32 для Win32:

```
.386
.model flat, stdcall
option casemap: none
include C:\masm32\include\windows.inc
include C:\masm32\include\kernel32.inc
includelib C:\masm32\lib\kernel32.lib

.data
<инициализированные данные>
...
.data?
<неинициализированные данные>
...
.const
<константы>
...
.code
<метка>:
      <текст программы>
```

```
    ...
    push NULL
    call ExitProcess
end <метка>
```

.386 – директива Ассемблера, определяющая набор команд процессора, который может быть использован в программе. Для приложений Win32 необходимо использовать эту директиву или выше, в зависимости от того, собираетесь ли вы использовать возможности, предоставляемые процессорами последующих поколений;

.model – директива Ассемблера, определяющая сегментную модель памяти приложения как плоскую (flat). Именно такая сегментная модель должна всегда использоваться при написании приложений для win32;

stdcall – говорит MASM32 о порядке передачи параметров. Согласно ему, данные передаются справа налево, но вызываемый ответственен за выравнивание стека. Платформа Win32 использует исключительно STDCALL;

option casemap: none – говорит MASM сделать метки чувствительными к регистрам, т. е. ExitProcess и exitprocess – это различные имена;

include – директива Ассемблера, добавляющая содержимое указанного после директивы файла в то место, где расположена директива. При использовании API-функций Win32 в программе должны быть описаны их прототипы. Это можно сделать вручную или путем подключения соответствующего *include*-файла. Подключаемые файлы находятся в директории *c:\masm32\include*. Файлы подключения имеют расширение *.inc* и прототипы функций DLL находятся в *.inc* файле с таким же именем, как и у этой DLL. Например, *ExitProcess* экспортируется *kernel32.lib*, так что прототип *ExitProcess* находится в *kernel32.inc*.

includelib – директива, сообщающая ассемблеру, какие библиотеки использует программа.

Например, сервис функции *ExitProcess* предоставляется *dll*-библиотекой *kernel32.dll*, следовательно, при сборке приложения необходимо подключить библиотеку импорта *kernel32.lib*.

Также можно указать имена библиотек импорта не с помощью директивы *.includelib*, а в командной строке при запуске линкера;

.data – директива, определяющая секцию инициализированных данных программы.

.data? – директива, определяющая секцию неинициализированных данных программы. Если нужно предварительно выделить некоторое количество памяти, но не инициализировать ее, то данная секция предназначается для этого. Преимущество неинициализированных данных следующее: они не занимают места в исполняемом файле. Например, если вы выделите 10 кб в секции *.data?*, *exe*-файл не увеличится на 10 кб. Вы всего лишь говорите компилятору, сколько места будет нужно, когда программа загрузится в память.

.const – директива, определяющая секцию констант. Константы не могут быть изменены программой.

Не обязательно задействовать все три секции. Объявляйте только те, которые хотите использовать.

.code – директива, определяющая секцию текста программы.

Все четыре директивы (секции) могут поделить адресное пространство на логические секции. Начало одной секции отмечает конец предыдущей. По факту есть две группы секций: данных и кода.

<метка>: – произвольная метка, устанавливающая границы кода. Обе метки должны быть идентичны. Весь текст программы должен располагаться между **<метка>:** и **end <метка>**.

push NULL – задание параметра функции ExitProcess.

call ExitProcess – вызов функции завершения приложения. Здесь код выхода NULL, но он может быть любым в пределах 32-разрядного целого числа. В полноценных приложениях нормой считается определение кода выхода в цикле обработки сообщений главного окна. Именно с помощью функции *ExitProcess* должно завершаться приложение Win32, написанное на Ассемблере.

Программу на языке Ассемблера для Win32 можно писать как в специализированной среде разработки (SASM, RadASM и др.), так и в обычном блокноте.

Для программирования в блокноте скачайте и установите программу Notepad++ (https://notepad-plus-plus.org/downloads/).

Скачиваем и распаковываем архив masm32v11r.zip по ссылке http://www.masm32.com/download.htm. Запускаем файл установщика *install.exe*. Выбираем для установки диск C. После установки на диске C появится папка *masm32*.

Для быстрого доступа из любой директории можно добавить пути в переменные среды Windows.

Для этого переходим: Этот компьютер -> Свойства -> Дополнительные параметры системы -> Переменные среды -> Системные переменные… Переменная Path… Изменить…

В конец списка добавляем строки:

```
C:\masm32
C:\masm32\bin
C:\masm32\help
C:\masm32\include
C:\masm32\lib
```

… сохраняем.

Создаем текстовый файл и переименовываем его в Test.asm.

Открываем файл в Notepad++ и пишем программу на языке Ассемблера, производящую вывод текстовой строки на экран консоли. Для вывода информации в консоль с использованием кириллического шрифта после открытия нового файла необходимо установить кодировку OEM 866, как показано на рис. 1.1.

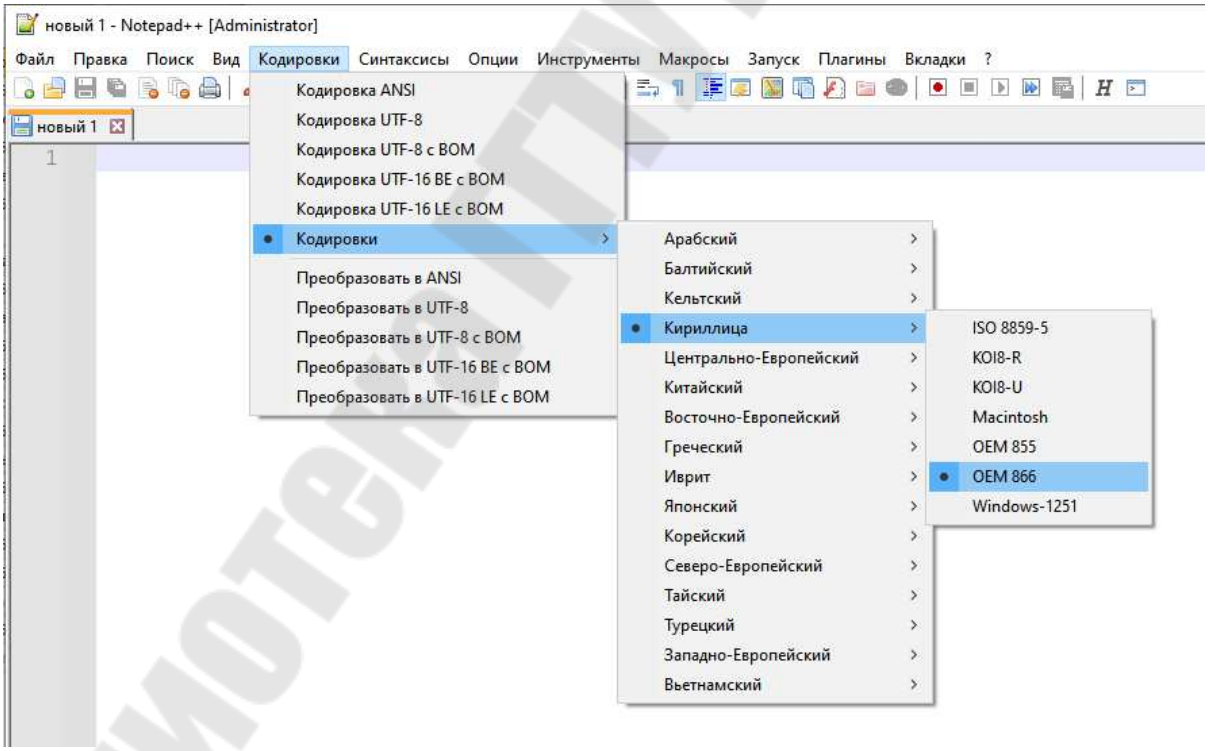


Рис. 1.1. Смена кодировки в Notepad++

Текст программы:

```
;-----------------------------------------
; Программа на masm32 для консоли
; (вывод сообщения)
;-----------------------------------------

.386
.model flat, stdcall
option casemap:none

; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\windows.inc
include C:\masm32\include\kernel32.inc
includelib C:\masm32\lib\kernel32.lib

; Сегмент данных
;-----------------------------------------
.data
string          db    "Здравствуй, Мир!", 0Ah, 0h
sConsoleTitle   db    "Мой первый проект", 0

; Сегмент кода
;-----------------------------------------
.code
start:
;-----------------------------------------
; Функция SetConsoleTitle устанавливает строку для заголовка
; текущей консоли
;
; ==== Аргументы ====
; lpConsoleTitle - указатель на строку для заголовка консоли

     push offset sConsoleTitle
     call SetConsoleTitle

; -----------------------------------------
; Функция GetStdHandle извлекает дескриптор для указанного
; стандартного устройства (стандартный ввод, стандартный вывод
; или стандартная ошибка).
; Функция возвращает дескриптор для указанного устройства.
;
; ==== Аргументы ====
```

```
; STD_INPUT_HANDLE - значение -10 стандартное устройство ввода
; STD_OUTPUT_HANDLE - значение -11 стандартное выходное
; устройство
; STD_ERROR_HANDLE - значение -12 устройство стандартных ошибок

      push STD_OUTPUT_HANDLE
      call GetStdHandle

; ----------------------------------------
; Функция WriteConsole записывает символьную строку в экранный
; буфер консоли, начинающийся с текущей позиции курсора.
;
; ==== Аргументы ====
; hConsoleOutput - дескриптор экранного буфера
; lpBuffer - указатель на массив символов (строку), которые будут
; записаны в экранный буфер консоли
; nNumberOfCharsToWrite  - число символов для записи (размер строки)
; lpNumberOfCharsWritten - указатель на переменную, которая
; принимает число фактических записей
; lpReserved - зарезервировано. MSDN рекомендует просто передать
; NULL

      push NULL
      push sizeof string
      push offset string
      push eax
      call WriteConsole

;----------------------------------------
; Функция Sleep приостанавливает работу по выполнению текущего
; потока на заданный промежуток времени.
;
; ==== Аргументы ====
; dwMilliseconds - длительность задержки [мс]. Значение
; БЕСКОНЕЧНО (INFINITE) вызывает бесконечную задержку

      push INFINITE
      call Sleep

;----------------------------------------
; Функция ExitProcess заканчивает работу процесса и всех его
; потоков
; ==== Аргументы ====
; uExitCode - возвращаемое значение
```

```
        push NULL
        call ExitProcess
end start
```

В папке с файлом *Test.asm* создаем текстовый файл *Run.bat* и добавляем в него в блокноте строку:

```
C:\masm32\bin\ml.exe /c /coff /Fl Test.asm
```

… сохраняем изменения.

Со значением ключей *ml.exe* можно ознакомиться здесь: https://wasm.in/blogs/kompiljacija-fajlov-asm-s-pomoschju-kompiljatora-ml-exe.74/.

После выполнения этой строки в папке появятся еще два файла *Test.lst* и *Test.obj*.

Добавим в файл *Run.bat* еще одну строку:

```
C:\masm32\bin\link.exe /SUBSYSTEM:CONSOLE /LIBPATH:C:\masm32\lib\
Test.obj
```

Подробную информацию о ключах link.exe можно посмотреть здесь: http://bitfry.narod.ru/link.htm.

После выполнения этой строки появится файл *Test.exe*.

Также в файл *Run.bat* добавьте строки (рис. 1.2):

```
pause
del Test.obj
start Test.exe
```

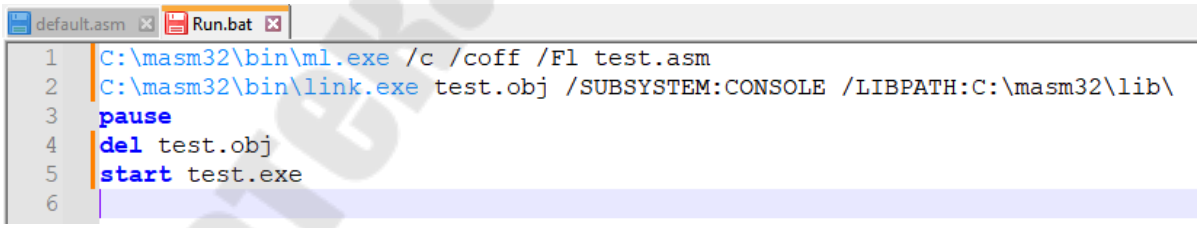


Рис. 1.2. Содержимое файла *Run.bat*

Сохраните изменения в файлах и запустите *Run.bat*. В результате отобразится следующее окно консоли (рис. 1.3).

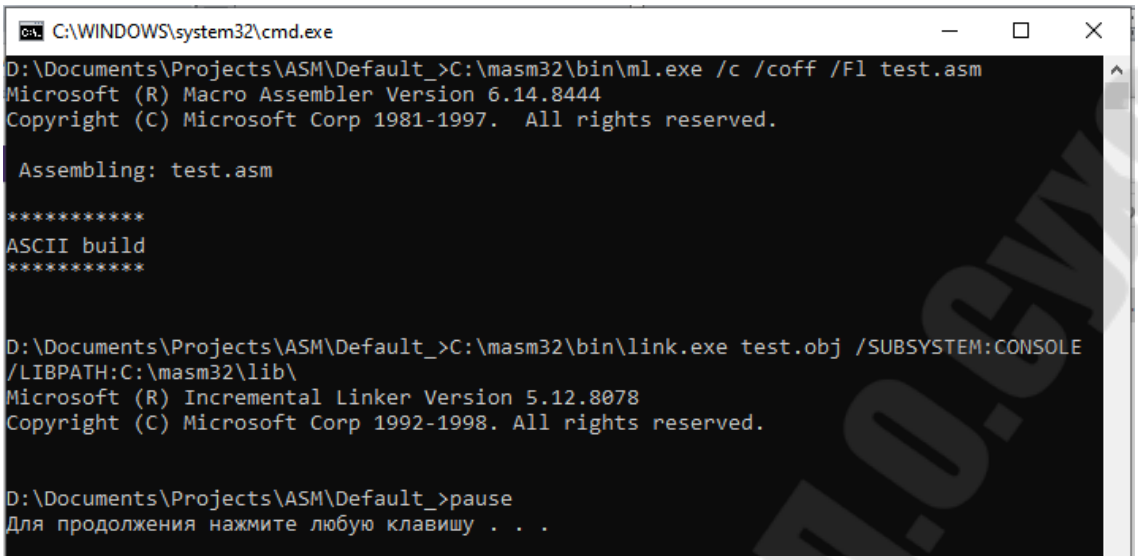


Рис. 1.3. Содержимое окна консоли

В результате выполнения файла *Test.exe* откроется консоль с заголовком «`Мой первый проект`» и содержанием «`Здравствуй, Мир!`» (рис. 1.4).

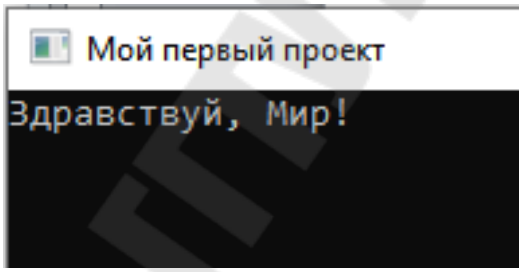


Рис. 1.4. Результат выполнения программы

Лабораторная работа № 1

Цель работы:

– знакомство с макроассемблером MASM32;

– создание минимального Windows-приложения на Ассемблере;

– изучение вызова функций API Win32 из ассемблерных приложений.

Задания для лабораторной работы

1. Набрать текст приведеной выше программы, произвести компиляцию и линковку, проверить работоспособность приложения.

2. Переписать приведенный выше текст программы с использованием макрокоманды **`invoke`**.

Макрокоманда `invoke` позволяет записывать вызовы в более удобном виде. Например, следующий вызов:

```
push MB_OKCANCEL
push offset hello_title
push offset hello_mess
push 0
call MessageBox
```

с помощью макрокоманды `invoke` будет записан следующим образом:

```
invoke MessageBox, 0, addr hello_mess, addr hello_title, MB_OKCANCEL
```

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Текст программ с комментариями.
5. Анализ результатов.
6. Выводы.

Контрольные вопросы

1. Что включает в себя соглашение о вызовах *stdcall*?
2. Как завершить выполнение программы на Ассемблере MASM32 и вернуться в операционную систему?
3. Какую функцию необходимо вызвать для вывода строки «`Здравствуй, Мир!`» на экран в программе на Ассемблере?
4. Какие аргументы необходимо передать функции вывода строки для вывода сообщения «`Здравствуй, Мир!`»?
5. Как объявить строку «`Здравствуй, Мир!`» в программе на Ассемблере?
6. Как передать адрес строки в качестве аргумента функции вывода строки?
7. Как объявить и использовать сегмент данных (data segment) в программе на Ассемблере?
8. Как объявить и использовать сегмент кода (code segment) в программе на Ассемблере?
9. Как объявить точку входа (entry point) программы на Ассемблере?
10. Как завершить выполнение программы на Ассемблере?

Глава 2. Типы данных Ассемблера. Пересылка данных

В MASM32 поддерживаются различные типы данных для объявления переменных. Вот некоторые из наиболее часто используемых типов данных в MASM32:

– **байт** (byte): используется для хранения целых чисел от –128 до 127 или беззнаковых чисел от 0 до 255;

– **двоичное слово** (word): используется для хранения целых чисел от –32 768 до 32 767 или беззнаковых чисел от 0 до 65 535;

– **двойное слово** (double word): используется для хранения целых чисел от –2 147 483 648 до 2 147 483 647 или беззнаковых чисел от 0 до 4 294 967 295;

– **учетверенное слово** (quad word): используется для хранения целых чисел от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 или беззнаковых чисел от 0 до 18 446 744 073 709 551 615;

– **массивы**: могут быть объявлены для хранения последовательности значений одного типа данных;

– **строки**: представляют последовательность символов и объявляются как массив байтов с завершающим нулевым байтом.

В общем случае все директивы объявления данных имеют такой синтаксис:

```
[имя] директива <значение>
```

Синтаксис параметра `<значение>` может быть следующим:

– `?` неинициализированные данные;

– `значение` начение элемента данных;

– `DUP (dup_выражение)` объявление и инициализация массивов.

К директивам объявления и инициализации простых данных относятся:

db (*Define Byte*) – определить байт. Директивой *db* можно задавать следующие значения:

– выражение или константу, принимающую значение из диапазона –128...+127 для чисел со знаком; 0...255 для чисел без знака;

– 8-битовое относительное выражение, использующее операции *HIGH* и *LOW*;

– символьную строку из одного или более символов, заключенную в кавычки.

dw (*Define Word*) – определить слово. Директивой *dw* можно задавать следующие значения:

– выражение или константу, принимающую значение из диапазона –32768...32767 для чисел со знаком; 0...65535 для чисел без знака;

– 1- или 2-байтовую строку, заключенную в кавычки.

dd (*Define Double word*) – определить двойное слово. Директивой *dd* можно задавать следующие значения:

– выражение или константу, принимающую значение из диапазона –2 147 483 648...+2 147 483 647 для чисел со знаком; для чисел 0...4 294 967 295 без знака;

– относительное или адресное выражение, состоящее из 16-битового адреса сегмента и 16-битового смещения;

– строку длиной до 4-х символов, заключенную в кавычки;

dq (*Define Quarter word*) – определить учетверенное слово;

df (*Define Far word*) – определить указатель дальнего слова;

dp (*Define Pointer*) – определить указатель 48 бит;

dt (*Define Ten Bytes*) – определить 10 байт.

Для резервирования памяти под массивы используется директива *dup*, например:

```
area   dw  10 dup(?)        ; резервируется  память объемом 10 слов
string db  20 dup('*')      ; строка заполняется кодом символа '*'
array  dw  3 dup(8)         ; массив из 3 слов инициализир. числом 8
t      db  4 dup(5 dup(8))  ; 20 восьмерок
```

Пример применения директив определения данных:

```
; определение байта
      v1    db    ?           ; не инициализировано
      v2    db    'ИП21'      ; символьная строка
      v3    db    6           ; десятичная константа
      v4    db    0afh        ; шестнадцатеричная константа
      v5    db    0110100b    ; двоичная константа
;определение слова
      w1    dw    bff3h       ; шестнадцатеричная константа
      w2    dw    11101111b   ; двоичная константа
      w3    dw    2,24,5,7    ; 4 константы
      w4    dw    23 dup(*)   ; 23 звездочки
;определение двойного слова

      d1    dd    ?           ; не определено
      d2    dd    'uAudf'     ; символьная строка
```

```
      d3    dd   08734       ; десятичная константа
      d4    dd   087h, 85fh  ; две константы
;определение учетверенного слова
      v1    dq   ?           ; не определено
      v2    dq   a83dh       ; константа
; определение массива слов
      mass  dw   100, -675, 54, 4556, 1
```

Пересылка данных

Команда MOV (*Move*) в MASM32 используется для копирования данных из одного места в другое. Она позволяет перемещать значения между регистрами процессора, памятью и другими местами хранения данных. Синтаксис команды MOV:

```
MOV [приемник], [источник]
```

С помощью этой команды можно переместить значение из источника в приемник. То есть команда MOV копирует содержимое источника и помещает это содержимое в приемник.

Никакие флаги при этом не изменяются.

При использовании этой команды следует учитывать, что имеются некоторые ограничения. А именно инструкция **MOV** не может:

– записывать данные в регистры *CS* и *IP*;

– копировать данные из одного сегментного регистра в другой сегментный регистр (сначала нужно скопировать данные в регистр общего назначения);

– копировать непосредственное значение в сегментный регистр (сначала нужно скопировать данные в регистр общего назначения).

Источником может быть один из следующих:

– область памяти (*mem*);

– регистр общего назначения (*reg*);

– непосредственное значение (например, число) (*imm*);

– сегментный регистр (*sreg*).

Приемником может быть один из следующих:

– область памяти (*mem*);

– регистр общего назначения (*reg*);

– сегментный регистр (*sreg*).

Пример использования инструкции MOV:

```
mov ax, 0B800h      ; установить ax = B800h
mov ds, ax          ; копировать значение из ax в ds
mov cl, 'a'         ; cl = 41h (ascii-код)
```

Просмотр выполнения программы в отладчике

Для отладки написанной программы можно воспользоваться отладчиком, например, *OllyDbg*. Программа *OllyDbg* — 32-битный отладчик (*debugger*), предназначенный для анализа и модификации откомпилированных исполняемых файлов и библиотек, работающих в режиме пользователя. *OllyDbg* отличается интуитивно понятным интерфейсом, подсветкой специфических структур кода, простотой в установке и запуске.

Исполняемый файл программы загружается в отладчик через меню: *File -> Open*.

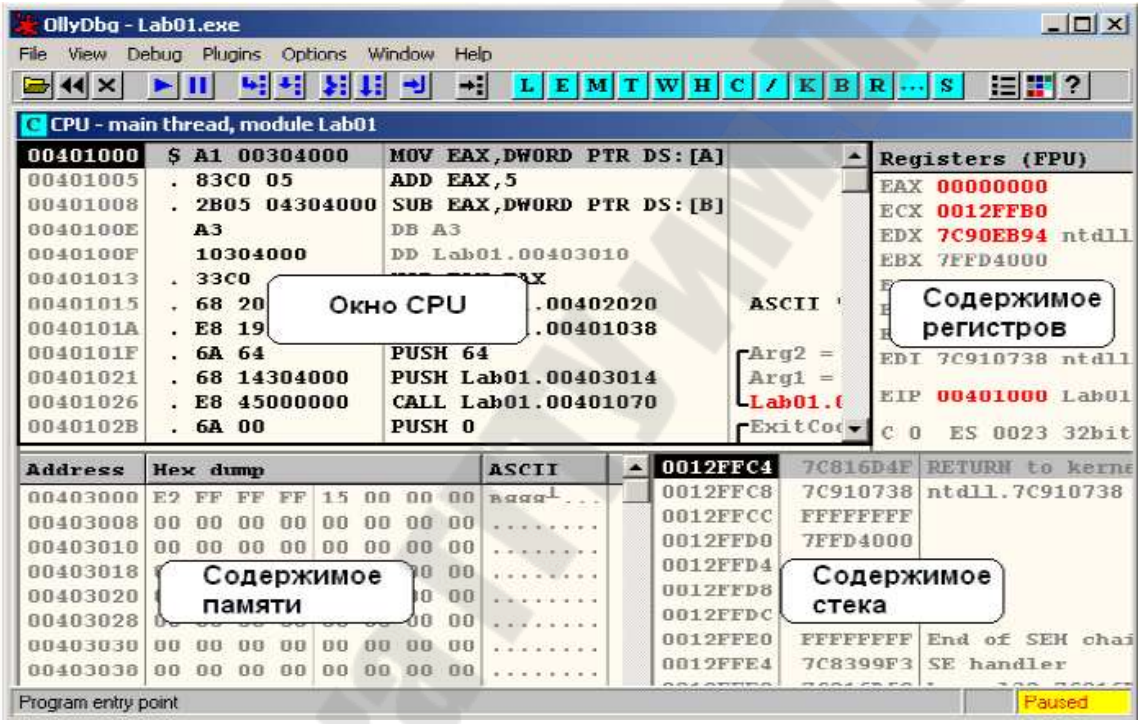


Рис. 2.1. Окно программы *OllyDbg*

В окне программы (рис. 2.1) можно выделить четыре области:

– окно CPU, в котором высвечиваются адреса команд исполняемой программы, шестнадцатеричные коды команд программы, результаты дисассемблирования и результаты выделения параметров процедур;

– окно регистров, в котором отображается содержимое регистров процессора, включая регистр флагов;

– окно памяти – окно, в котором высвечиваются адреса памяти (*Address*) и ее шестнадцатеричный (*Hex dump*) и символьный (ASCII) дампы;

– окно стека, в котором отображается содержимое области памяти, отведенной под стек.

Стек – область памяти, в которой хранится информация о каждом обращении к функции. Кроме того, в стеке могут временно храниться данные, например, содержимое регистров.

Стек увеличивается в сторону уменьшения адресов памяти. Когда вы добавляете что-то в стек, то эти данные помещаются по адресу памяти младшему, чем адрес предыдущего элемента. То есть по мере роста стека адрес вершины стека уменьшается.

Инструкция *push* заносит что-нибудь на верх стека, а *pop* уносит данные оттуда. Например, *push EAX* выделяет место наверху стека и помещает туда значение из регистра *EAX*, а *pop EAX* переносит любые данные из верхней части стека в *EAX* и освобождает эту область памяти.

Регистр *ESP* — указатель стека. Он хранит адрес вершины стека и изменяет свое значение при записи или чтении из стека.

В исходный момент времени курсор находится в окне CPU, т. е. программа готова к выполнению. Для выполнения команд программы в пошаговом режиме используют следующие функциональные клавиши отладчика:

– F7 (*step into*) – выполнить шаг с заходом в тело процедуры;

– F8 (*step over*) – выполнить шаг, не заходя в тело процедуры.

Адрес начала программы *00401000h*.

Адрес начала раздела инициированных данных – *00403000h*. Каждое значение занимает столько байт, сколько резервируется соответствующей директивой. В отладчике каждый байт представлен двумя шестнадцатеричными цифрами. Кроме того, используется обратный порядок байт, т. е. младший байт числа находится в младших адресах памяти (перед старшим). Если шестнадцатеричная комбинация соответствует коду символа, то он высвечивается в следующем столбце (ASCII), иначе в нем высвечивается точка.

Неинициированные данные располагаются после инициированных с адреса, кратного 16.

При каждом выполнении команды мы можем наблюдать изменение данных в памяти и/или в регистрах, отслеживая процесс выполнения программы и контролируя правильность промежуточных результатов.

Так, после выполнения первой команды число *A* копируется в регистр *EAX*, который при этом подсвечивается красным. При записи в регистр порядок байт меняется на прямой, при котором первым записан старший байт.

Лабораторная работа № 2

Цель работы:

– знакомство с типами переменных Ассемблера и их размещением в памяти;

– изучение команд пересылки данных Ассемблера;

– получение навыков работы с отладчиком.

Задания для лабораторной работы

1. В соответствии с вариантом задания (табл. 2.1) написать программу, где в сегменте данных будут созданы переменные *a*, *b*, массив данных и строка. Произвести пересылку данных в указанное место. Произвести обмен значений переменных через стек.

Таблица 2.1

Задание к лабораторной работе № 2

Номер	Переменные				Место пересылки	
	a	***b***	***m***	***s***	***a***	***b***
1	–34	65465	5,5,5,5,5,5,5	Привет	*EAX*	*EDX*
2	–5	0x56	5,89,65,9	Яблоко	*EBX*	*EAX*
3	45	765765	0,0,0,0,0,0,1	Книга	*ECX*	*EBX*
4	0*b*1011	0x76*A*	765765, –1,0,0, 99999, 56678945	Солнце	*EDX*	*ECX*
5	–56	–67565765	–675,765, –5643,789, –765,777	Дом	*EAX*	*EDX*
6	0*b*111	–39879798	9,8,7,9,8,7	Музей	*EBX*	*ECX*
7	87687	–76	34,34,34,34	Квартира	*ECX*	*EAX*
8	67	0*b*1001	56,78,65,0,1	Медведь	*EDX*	*EBX*
9	0*b*110011	–5765	543543, 7675, 675476904, 56	Собака	*EAX*	*ECX*

Продолжение табл. 2.1

Номер	Переменные				Место пересылки	
	a	***b***	***m***	***s***	***a***	***b***
10	0x45	11111*b*	654,98, –432,–4322	Ручка	*EBX*	*EDX*
11	–2	7765765	–9,–2,8,67,5	Радуга	*ECX*	*EAX*
12	–1	337865765	0x543, 0x34, 0x7647*FFF*, 0x5633	Звезда	*EDX*	*EAX*
13	0x*FF*095	87687	–9,–9,–9,–9,–9	Часы	*EAX*	*EDX*
14	0*b*100011	6754	44,44,44,44	Минуты	*EBX*	*EAX*
15	0x5*F*	0x*AB*54	0x*ADFFFF*, 0x34, 0x*DF* 0x45, 0x67	Планета	*ECX*	*EBX*
16	0x34*F*	0*b*1111111	56,78,689, 342436,6	Дождь	*EDX*	*ECX*
17	0x78653	–4	3,3,3,3,3,3,3	Чашка	*EAX*	*EDX*
18	99	–2376575	0,0,0,0,0,0,0	Футбол	*EBX*	*EAX*
19	909	547659099	0,1,2,3,0,1, 2,3	Чемодан	*ECX*	*EDX*
20	0*b*11	787423240	0*b*1111, 0*b*10101, 0*b*10000, 0*b*101111, 0*b*1111111	Парк	*EDX*	*EAX*
21	0x78*FF*1*F*	–6534	0x45, 0x5, 0x*F*4545, 0x6, 0x*FFF*	Кофе	*EAX*	*EBX*
22	0*b*1111111	0x*DF*	55,55,55,55	Луна	*EBX*	*ECX*
23	0x567*FF*	76576	11110000, 111000, 11110000, 111000	Птица	*ECX*	*EDX*
24	–8798675	543	0x*FF*00*FF*, 0x*FF*00*FF*, 0x*FF*00*FF*, 0x*FF*00*FF*	Океан	*EDX*	*EAX*

Окончание табл. 2.1

Номер	Переменные				Место пересылки	
	a	***b***	***m***	***s***	***a***	***b***
25	–999999	–22	–65,–76575, 0,5423543	Музыка	EAX	EBX
26	7777777	0x*ABCD*	0*b*1110001, 0x*FFFFFF*, 675765, –4	Компьютер	*EBX*	*ECX*
27	–54765899	0*b*11101	555,6667, 77777,8888	Шоколад	*ECX*	*EDX*
28	–34	–88945765	564,–1,1,0, 45,567890	Кинотеатр	*EDX*	*EAX*
29	3	0x*FADFF*	0,0,1,0,0,1	Машина	*EAX*	*EBX*
30	0x5435*D*	0x111111		Дворец	*EBX*	*ECX*

Для создания переменных выделить минимально необходимый объем памяти.

2. Открыть программу в отладчике. Указать адреса расположения переменных в памяти, заполнив табл. 2.2.

Таблица 2.2

Адреса переменных в памяти

Имя переменной	Адрес	Расположение байт в памяти

3. Выполнить программу в отладчике в пошаговом режиме. После выполнения каждого шага заносить данные в табл. 2.3.

Таблица 2.3

Состояние регистров* и стека

EAX	***EBX***	***ECX***	***EDX***	***ESP***	***Стек***

*Выбрать необходимое.

4. По результатам п. 2 и 3 сделать выводы.

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Текст программ с комментариями.
5. Результаты выполнения программы в отладчике.
6. Выводы.

Контрольные вопросы

1. Какой тип данных в MASM32 используется для хранения целых чисел размером 8, 16, 32 бита?
2. Какая команда в MASM32 используется для пересылки значения из одного регистра в другой?
3. Какой тип данных в MASM32 используется для хранения символов?
4. Какая команда в MASM32 используется для пересылки значения из регистра в стек?
5. Какая команда в MASM32 используется для пересылки значения из стека в регистр?
6. Опишите работу в отдадчике OllyDbg.

Глава 3. Функции ввода/вывода. Логические и арифметические команды. Программирование линейных алгоритмов

Функции и процедуры ввода, вывода

В макроассемблере MASM32 для ввода/вывода информации можно воспользоваться:

– функциями **API Windows**;

– процедурами (функциями) библиотеки **masm32.lib**.

ОС Windows предоставляет огромное количество ресурсов Windows-программам через **Windows API** (Application Programming Interface) – коллекцию полезных функций, располагающихся непосредственно в операционной системе и готовых для использования программами. Эти функции находятся в нескольких динамически подгружаемых библиотеках (DLL), таких как *kernel32.dll*, *user32.dll* и *gdi32.dll*.

Kernel32.dll содержит API функции, взаимодействующие с памятью и управляющие процессами, *user32.dll* контролирует пользовательский интерфейс, *gdi32.dll* отвечает за графические операции.

Существуют также другие библиотеки. Код API-функций не включается в исполняемый файл.

Вызов системных функций API Win32 из программы на ассемблере подчиняется набору соглашений *stdcall*. Эти соглашения заключаются в следующем:

– **регистр символов в имени функции не имеет значения**. Например, функции с именами *ExitProcess* и *exitprocess* – это одна и та же функция;

– **аргументы передаются вызываемой функции через стек**. Если аргумент укладывается в 32-битное значение и не подлежит модификации вызываемой функцией, он обычно записывается в стек непосредственно. В остальных случаях программист должен разместить значение аргумента в памяти, а в стек записать 32-битный указатель на него. Таким образом, все передаваемые функции API параметры представляются 32-битными величинами, и количество байт, занимаемых в стеке для передачи аргументов, кратно четырем;

– вызывающая программа загружает аргументы в стек последовательно, начиная с последнего, указанного в описании функции, и заканчивая первым. После загрузки всех аргументов **программа вызывает функцию командой *call***;

– за возвращение стека в исходное состояние после возврата из функции API отвечает сама эта вызываемая функция. Программисту заботиться о восстановлении указателя стека *esp* нет необходимости;

– вызываемая функция API гарантированно сохраняет регистры общего назначения *ebp*, *esi*, *edi*. **Регистр *eax*, как правило, содержит возвращаемое значение.** Состояние остальных регистров после возврата из функции API следует считать неопределенным.

Для того чтобы в программе MASM32 воспользоваться функциями API Windows, необходимо в начале программы добавить следующие подключаемые файлы (**.inc*) и библиотеки импорта (**.lib*):

```
include C:\masm32\include\windows.inc
include C:\masm32\include\kernel32.inc
include C:\masm32\include\user32.inc
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\user32.lib
```

MASM32.LIB – это набор процедур для обеспечения дополнительной функциональности проекта MASM32. Каждая процедура была написана как отдельный модуль. Полный исходный код библиотеки MASM32 находится в каталоге M32LIB в MASM32.

Чтобы использовать эти процедуры, программа должна включать следующие две строки:

```
include C:\masm32\include\masm32.inc
includelib C:\masm32\lib\masm32.lib
```

Для выполнения операций над цифровыми данными необходимо придерживаться следующей последовательности действий:

1. Произвести ввод цифровой информации. Ввод данных может быть выполнен с помощью функций **`ReadConsole, StdIn`**.

2. Выполнить подготовку полученной информации для дальнейшей обработки.

Во-первых, функции ввода при приеме строки, как правило, дополняют ее «лишними» символами переноса и возврата каретки (***`0Dh, 0Ah`***).

Во-вторых, принятые через устройство ввода данные, представляют собой, по сути, текстовую, а не цифровую информацию. Функции ввода данных воспринимают вводимые цифры как текст в виде ASCII-кодов. Например, если на клавиатуре набрать число «537», в память запишется строка «35 33 37 0*D* 0*A* 00», где «35», «33», «37» – коды цифр 5, 3, 7, а не сами значения, а коды «0*D*», «0*A*», «00» – символы возврата каретки, перевода строки и окончания строки.

Поэтому перед началом обработки принятую строку символов необходимо очистить от «лишних» данных (убрать «0*D* 0*A* 00»), а затем оставшуюся информацию («35 33 37») перевести в двоичный код числа 537.

Подготовку данных можно выполнить с помощью функций **`StripLF, atodw, atol, ustr2dw, StrToFloat`**.

3. Выполнить обработку информации. Под обработкой информации понимается выполнение арифметических, логических и прочих операций вычислительного характера, в результате которых находится решение поставленной задачи.

4. Выполнить подготовку обработанной информации для вывода. Суть данной операции состоит в том, чтобы из цифровой информации получить текстовую строку ASCII-кодов для вывода на экран. Для этого можно воспользоваться функциями **`dwtoa, ltoa, udw2str, Float2Str`**.

5. Произвести вывод обработанной информации. Полученную текстовую строку можно вывести с помощью следующих функций: **`WriteConsole, WriteFile, MessageBox, StdOut`**.

Функции ввода/вывода API Windows

1. Функция **SetConsoleTitle** устанавливает строку заголовка для текущего консольного окна.

Синтаксис:

```
BOOL SetConsoleTitle (PCTSTR lpConsoleTitle);
```

Параметры:

– **lpConsoleTitle** – указатель на строку с символом нуля в конце, содержащую текст, который будет показан на экране в заголовке консольного окна. Общий размер ее должен быть меньше чем 64КБ.

Тип **PCTSTR** можно понять так:

P – *Pointer* (указатель);

C – *Constant* (константа);

T – *TCHAR*;

STR – *String* (строка).

Чтобы использовать SetConsoleTitle в MASM32, необходимо выполнить включить файл windows.inc в начале кода на ассемблере.

Пример:

```
include C:\masm32\include\windows.inc

.data
sConsoleTitle  BYTE  "Console Title", 0
…
.code
…
     push OFFSET sConsoleTitle
     call SetConsoleTitle
```

2. Функция **GetStdHandle** извлекает дескриптор для указанного стандартного устройства.

Синтаксис:

```
HANDLE GetStdHandle (DWORD nStdHandle);
```

Параметры:

– nStdHandle – стандартное устройство, для которого дескриптор должен быть возвращен. Этот параметр может быть одним из следующих значений:

– STD_INPUT_HANDLE – дескриптор стандартного устройства ввода данных (дескриптор консольного буфера ввода);

– STD_OUTPUT_HANDLE – дескриптор устройства стандартного вывода (дескриптор активного экранного буфера консоли);

– STD_ERROR_HANDLE – дескриптор стандартной ошибки устройства (дескриптор активного экранного буфера консоли).

Функция возвращает **дескриптор** для указанного устройства в ***EAX***. Тип HANDLE означает дескриптор объекта. Дескриптор – 32-битное беззнаковое целое число.

Стандартные дескрипторы процесса определяются конфигурацией флага /SUBSYSTEM, переданного компоновщику при создании приложения. Если указать /SUBSYSTEM:CONSOLE, операционная система получит запрос на заполнение дескрипторов данными сеанса консоли при запуске, если родительский объект еще не заполнил стандартную таблицу дескрипторов путем наследования. /SUBSYSTEM:WINDOWS, в свою очередь, означает, что приложению не нужна консоль и, скорее всего, не будут использоваться стандартные дескрипторы.

Если функция не удалась, возвращается специальное значение INVALID_HANDLE_VALUE.

Пример:

```
.data?
     hStdout  DWORD ?
     cWritten DWORD ?
…
.code
     push STD_OUTPUT_HANDLE
     call GetStdHandle

     mov hStdout, eax
     mov cWritten, ebx
```

3. Функция **WriteConsole** из API Windows позволяет программе выводить данные на консольный экран, начиная с текущего положения курсора.

Синтаксис:

```
BOOL WriteConsole (HANDLE      hConsoleOutput,
                   CONST VOID* lpBuffer,
                   DWORD       nNumberOfCharsToWrite,
                   LPDWORD     lpNumberOfCharsWritten,
                   LPVOID      lpReserved);
```

Параметры:

– `hConsoleOutput` – дескриптор буфера экрана консоли;

– `lpBuffer` – указатель на буфер, содержащий символы для записи в буфер экрана консоли;

– `nNumberOfCharsToWrite` – количество записываемых символов;

– `lpNumberOfCharsWritten` – указатель на переменную, которая получает количество фактически записанных символов;

– `lpReserved` – зарезервировано, **должно быть** значение `NULL`.

Функция `WriteConsole` возвращает значение `TRUE` в случае успешного вывода или `FALSE` в случае ошибки.

Пример:

```
.data
string    BYTE   ‘Программа на ассемблере’, 0
     …
.data?
hStdout   DWORD  ?
cWritten  DWORD  ?
     …
.code
     …
; Получение дескриптора консольного вывода
     push STD_OUTPUT_HANDLE
     call GetStdHandle
     mov hStdout, eax
     …
; Вывод данных в консоль
     push NULL
     push OFFSET cWritten
     push SIZEOF string
     push OFFSET string
     push hStdout
     call WriteConsole
```

4. Функция **`ReadConsole`** из API Windows позволяет считывать ввод с консоли в MASM32. Она позволяет программе получить введенные пользователем данные с клавиатуры.

Синтаксис:

```
OOL ReadConsole (HANDLE  hConsoleInput ,
                 LPVOID  lpBuffer,
                 DWORD   nNumberOfCharsToRead,
                 LPDWORD lpNumberOfCharsRead ,
                 LPVOID  lpReserved);
```

Параметры:

– hConsoleInput – дескриптор стандартного устройства ввода консоли;

– lpBuffer – указатель на буфер, который получает данные, считываемые из входного буфера консоли;

– nNumberOfCharsToRead – число считываемых символов;

– lpNumberOfCharsWritten – указатель на переменную, которая получает количество фактически считанных символов;

– lpReserved – зарезервировано, **должно быть** значение NULL.

Вводимая строка дополняется символами 0Dh, 0Ah, 0.

Функция завершается при нажатии клавиши *Enter*.

Функция возвращает значение TRUE в случае успешного считывания или FALSE в случае ошибки.

Пример:

```
.data?
      hStdin     DWORD  ?
      cWritten   DWORD  ?
      strInput   BYTE       20 DUP(?)
      …
.code
            …
; Получение дескриптора консольного ввода
      push STD_INPUT_HANDLE
      call GetStdHandle
      mov hStdin, eax

; Чтение данных из консоли
      push NULL
      push OFFSET cWritten
      push SIZEOF strInput
      push OFFSET strInput
      push hStdin
      call ReadConsole
```

5. Функция **WriteFile** (fileapi.h) записывает данные в указанный файл или устройство ввода-вывода (ввода-вывода).

Синтаксис:

```
BOOL WriteFile (HANDLE       hFile,
                LPCVOID      lpBuffer,
                DWORD        nNumberOfBytesToWrite,
```

```
    LPDWORD      lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped);
```

Параметры:

– `hFile` – дескриптор устройства ввода-вывода или файла (например, файл, физический диск, буфер консоли);

– `lpBuffer` – указатель на буфер, содержащий данные, записываемые в файл или устройство;

– `nNumberOfBytesToWrite` – количество байтов, записываемых в файл или устройство;

– `lpNumberOfBytesWritten` – указатель на переменную, которая получает количество байтов, записанных при использовании синхронного параметра `hFile`;

– `lpOverlapped` – указатель на структуру `OVERLAPPED` требуется, если параметр `hFile` был открыт с `FILE_FLAG_OVERLAPPED`, в противном случае этот параметр может иметь значение `NULL`.

Типы данных *Windows*:

`LPCVOID` – указатель на константу любого типа (*typedef CONST void *LPCVOID*);

`LPDWORD` – указатель на *DWORD* (*typedef DWORD *LPDWORD*).

Пример:

```
.data
string    BYTE   ‘Программа на ассемблере’, 0
     …
.data?
hStdout   DWORD  ?
cWritten  DWORD  ?
     …
.code
     …
; Получение дескриптора консольного вывода
     push STD_OUTPUT_HANDLE
     call GetStdHandle
     mov hStdout, eax
     …
; Вывод данных в консоль
     push NULL
     push OFFSET cWritten
     push SIZEOF string
     push OFFSET string
     push hStdout
     call WriteFile
```

6. Функция **MessageBox** из API Windows позволяет программе выводить диалоговое окно сообщения в MASM32.

Синтаксис:

```
int MessageBox(HWND     hWnd,
               LPCTSTR  lpText,
               LPCTSTR  lpCaption,
               UINT     uType);
```

Параметры:

– hWnd – дескриптор родительского окна. Дескриптор можно считать числом, представляющим окно, к которому вы обращаетесь;

– lpText – это указатель на текст, отображаемый в клиентской части окна сообщения;

– lpCaption – заголовок диалогового окна. Если этот параметр имеет значение NULL, заголовок по умолчанию – Error;

– uType – устанавливает иконку, число и вид кнопок окна.

В Win32 HWND, LPCSTR и UINT имеют разрядность 32 бит.

Функции ввода/вывода библиотеки *masm32.lib*

7. Функция **StdIn** получает текстовый ввод с консоли и помещает его в буфер, указанный в качестве параметра. Функция завершается при нажатии клавиши *Enter*.

Синтаксис:

```
StdIn PROC, lpszBuffer:DWORD, bLen:DWORD
```

Параметры:

– lpszBuffer – адрес буфера для приема текстового ввода с консоли;

– bLen – длина буфера.

Консоль определяет длину максимального размера буфера 128 байт. Положение ввода на консоли может быть задано с помощью функции **locate**.

Пример:

```
.data?
    buffer   BYTE    20 DUP(?)
…
.code
…
    push LENGTHOF buffer        ; bLen
```

```
push OFFSET buffer          ; lpszBuffer
call StdIn
```

8. Функция **StdOut** (чувствительна к регистру) выводит в консоль нуль-терминированную строку в текущей позиции.

Синтаксис:

```
StdOut PROC, lpszText:DWORD
```

Параметры:

– lpszText – нуль-терминированная строка.

Ограничение на длину отображаемой строки отсутствует.

Позицию отображаемого текста на консоли можно задать с помощью функции **locate**.

Пример:

```
.data
promptText   db  "Введите значение A:", 0Dh, 0Ah, 0
…
.code
      push OFFSET promptText
      call StdOut
```

Функции подготовки и преобразования данных библиотека *masm32.lib*

1. Функция **StripLF** предназначена для удаления CRLF (ASCII 0Dh,0Ah,) путем записи ASCII-кода нуля на место первого ASCII-кода 0Dh.

Синтаксис:

```
StripLF PROC strng:DWORD
```

Параметры:

– strng – адрес буфера, который необходимо усечь.

Процедура разработана для ввода в консольном режиме, где к концу результата добавляется ASCII 13,10. Это влияет на ввод числовых значений, которые преобразуются из строки в DWORD. Удаление CRLF позволяет правильно преобразовать введенные числа.

Пример:

```
.data?
      string     BYTE      20 DUP(?)
…
```

```
.code
…
; Ввод данных
      push LENGTHOF string
      push OFFSET string
      call StdIn
…
; Удаление символов конца строки
      push OFFSET string
      call StripLF
```

2. Функция **atodw** преобразует десятичную строку в DWORD.

Синтаксис:

```
atodw PROC USES EDI ESI, String:PTR BYTE
```

Параметры:

– String – адрес десятичной строки для преобразования.

Параметр String является адресом (указателем) размера *DWORD*.

Возвращаемое значение: значение *DWORD* возвращается в *EAX*.

Пример:

```
.data?
string  BYTE  20 DUP(?)
      …
.code
      …
; Ввод данных
      push LENGTHOF string
      push OFFSET string
      call StdIn

; Удаление символов конца строки
      push OFFSET string
      call StripLF

; Преобразование строки в число
      push OFFSET string
      call atodw
```

3. Функция **dwtoa** преобразует DWORD в строку.

Синтаксис:

```
dwtoa PROC PUBLIC USES ESI EDI, dwValue:DWORD, lpBuffer:DWORD
```

Параметры:

– dwValue – преобразуемая переменная типа DWORD;

– lpBuffer – указатель на строку.

Пример:

```
.data?
      resStr  BYTE   16 DUP(?)
      X       DWORD  ?
…
.code
…
      mov X, AX

; Преобразование числа в строку
      push OFFSET resStr
      push X
      call dwtoa
```

4. Функция **ustr2dw** преобразует нуль-терминированную строку в UNSIGNED DWORD.

Синтаксис:

```
ustr2dw PROC pszString:DWORD
```

Параметры:

– pszString – адрес нуль-терминированной строки для преобразования.

Возвращаемое значение: значение UNSIGNED DWORD возвращается в EAX.

Пример:

```
.data?
string     BYTE      20 DUP(?)
…
.code
           …
; Ввод данных
      push LENGTHOF string
      push OFFSET string
      call StdIn

; Удаление символов конца строки
      push OFFSET string
```

```
        call StripLF

; Преобразование строки в число
        push OFFSET string
        call ustr2dw
```

5. Функция **udw2str** преобразует UNSIGNED DWORD в строку.

Синтаксис:

```
udw2str PROC dwNumber:DWORD, pszString:DWORD
```

Параметры:

– dwNumber – беззнаковое число DWORD для преобразования;

– pszString – буфер для получения результата.

Пример:

```
.data?
resStr  BYTE   16 DUP(?)
X       DWORD  ?
…
.code
…
        mov X, AX

; Преобразование числа в строку
        push OFFSET resStr
        push X
        call udw2str
```

6. Функция **atol** преобразует числовую нуль-терминированную строку в 32-битное знаковое целое число.

Синтаксис:

```
atol PROC lpSrc:DWORD
```

Параметры:

– lpSrc – адрес нуль-терминированной строки.

Возвращаемое значение – целое 32-битное число со знаком.

Алгоритм завершается на первом символе меньше *30h* (символ «0»).

Алгоритм удаляет все пробелы, но если он используется в среде, где присутствуют ведущие табуляции. Табуляции должны быть удалены в первую очередь.

Алгоритм принимает знаки «+» и «–» и рассматривает числовую строку без знака как положительное целое число.

За ведущим символом знака не должен следовать пробел, иначе процедура завершится ошибкой.

Пример:

```
.data?
string   BYTE  20 DUP(?)
…
.code
            …
; Ввод данных
      push LENGTHOF string
      push OFFSET string
      call StdIn

; Удаление символов конца строки
      push OFFSET string
      call StripLF

; Преобразование строки в число
      push OFFSET string
      call atol
```

7. Функция **ltoa** преобразует 32-битное знаковое целое число (LONG) в нуль-терминированную строку.

Синтаксис:

ltoa PROC **lValue:**DWORD**, lpBuffer:**DWORD

Параметры:

– `lValue` – преобразуемое 32-битное знаковое целое число;

– `lpBuffer` – указатель на нуль-терминированную строку необходимой длины.

Эта процедура основана на функции «wsprintfA» Windows API. Буфер должен быть достаточно длинным, чтобы получить результат максимальной длины. Рекомендуемый размер буфера – 16 байт для целей размера и выравнивания.

Пример:

```
.data
      resStr  BYTE   16 DUP(?)
      …
```

```
.data?
      X       DWORD  ?
      …
.code
      …
      mov X, AX

; Преобразование числа в строку
      push OFFSET resStr
      push X
      call ltoa
```

Логические и арифметические команды MASM32

Ассемблер MASM32 предоставляет инструкции (команды) для выполнения логических операций над данными. Вот некоторые из них.

Команда **AND** используется для выполнения побитовой операции «логическое И» над двумя операндами `op1` и `op2` и сохранения результата на месте первого операнда. Каждый бит результата будет равен «1» только в том случае, если соответствующие биты обоих операндов равны «1». В противном случае бит результата будет равен «0». Синтаксис команды **AND**:

```
AND op1, op2
```

Варианты команды **AND**:

```
AND reg, reg        AND mem, imm        AND reg, imm
AND reg, mem        AND mem, reg
```

где *reg* – регистр;
mem – память;
imm – непосредственное число.

Пример сброса бит 7:4 (биты 3:0 не меняются):

```
mov al, 00111011b        ; AL = 0011 1011b
and al, 00001111b        ; AL = 0000 1011b
```

Команда **OR** используется для выполнения побитовой операции «логическое ИЛИ» над двумя операндами и сохранения результата на месте первого операнда. Каждый бит результата будет равен «1», если

хотя бы один из соответствующих битов операндов равен «1». Синтаксис команды **OR**:

```
OR op1, op2
```

Варианты команды **OR**:

```
OR reg, reg          OR mem, imm          OR reg, imm
OR reg, mem          OR mem, reg
```

Пример получения ASCII-кода символа «5», равного 35h:

```
mov dl, 5                ; DL = 0000 0101b (число 5)
or dl, 30h               ; DL = 0011 0101b (код числа 5 = 35h)
```

Команда **XOR** используется для выполнения побитовой операции «исключающее ИЛИ» над двумя операндами и сохранения результата на месте первого операнда. Каждый бит результата будет равен «1» только в том случае, если соответствующие биты операндов различны. Если оба бита равны «0» или равны «1», то бит результата будет равен «0». Синтаксис команды **XOR**:

```
XOR op1, op2
```

Варианты команды **XOR**:

```
XOR reg, reg         XOR mem, imm         XOR reg, imm
XOR reg, mem         XOR mem, reg
```

Пример очистки регистра с помощью команды XOR:

```
mov dl, 5                ; DL = 0000 0101b
xor dl, dl               ; DL = 0000 0000b
```

Команда **NOT** используется для выполнения побитовой операции «логическое НЕ» над одним операндом. Каждый бит результата будет инвертирован: если бит операнда равен «0», то бит результата будет равен «1», и наоборот. Синтаксис команды **NOT**:

```
NOT op1
```

Варианты команды **NOT**:

```
NOT reg
NOT mem
```

Например, **обратный код** числа F0h равен 0Fh:

```
mov al, 11110000b              ; AL = 1111 0000b (F0h)
not al                         ; AL = 0000 1111b (0Fh)
```

Команды **AND, OR, XOR** всегда сбрасывают флаги переполнения (*OF*) и переноса (*CF*). Кроме того, они устанавливают значения флагов знака (*SF*), нуля (*ZF*) и четности (*PF*) в соответствии со значением результата.

Команда **NOT** не изменяет флаги процессора.

Команда **ADD** выполняет арифметическое сложение двух операндов. Операнды должны иметь одинаковый размер. Результат записывается на место операнда 1.

```
ADD op1, op2
op1 = op1 + op2
```

Варианты команды ADD:

```
ADD reg, reg        ADD reg, mem        ADD mem, imm
ADD reg, imm        ADD mem, reg
```

Пример выполнения:

```
a dd 45                ; задание типа и значения переменной A
b dd -32               ; задание типа и значения переменной B
e dd ?                 ; задание типа переменной E
mov eax, a             ; копирование A в регистр EAX
add eax, b             ; сложение регистра EAX с переменной B
mov e, eax             ; копирование суммы из EAX в E
```

Команда **ADC** (*Add with Carry*) производит целочисленное сложение двух знаковых или беззнаковых операндов и флага переноса. При этом оба операнда одновременно не могут быть переменными в памяти.

```
ADC op1, op2
op1 = op1 + op2 + флаг CF
```

Команда ADC обычно используется в многобайтных или многословных (*multi-word*) операциях сложения:

```
xor edx, edx              ; EDX = 0
mov eax, 0FFFF FFFFh      ; EAX = 4 294 967 295
add eax, 0FFFF FFFFh      ; EAX = EAX + 4 294 967 295
adc edx, 0                ; EDX = EDX + CF  (учитываем перенос)
                          ; EDX:EAX = 0000 0001h:FFFF FFFEh
```

Команда **SUB** выполняет вычитание двух операндов. Операнды должны иметь одинаковый размер. Результат записывается на место операнда 1.

```
SUB op1, op2
op1 = op1 - op2
```

Варианты команды SUB:

```
SUB reg, reg        SUB reg, mem        SUB mem, imm
SUB reg, imm        SUB mem, reg
```

Для умножения чисел без знака предназначена команда **MUL** (*Multiply*). У этой команды только один операнд – второй множитель, который должен находиться в регистре или в памяти. Местоположение первого множителя и результата задается неявно и зависит от размера операнда (табл. 3.1).

Таблица 3.1

Местоположение первого множителя и результата

Размер операнда	1-й множитель	Результат
1 байт	*AL*	*AX*
2 байта	*AX*	*DX:AX*
4 байта	*EAX*	*EDX:EAX*

Запись «*DX:AX*» означает, что старшее слово результата будет находиться в *DX*, а младшее – в *AX*.

Пример:

```
mul bl                    ; AX = AL * BL
mul ax                    ; DX:AX = AX * AX
```

Команда **IMUL** умножает целые числа со знаком и может использовать один, два или три операнда:

– один операнд, аналогично команде MUL;
– два операнда:

```
IMUL op1, op2
op1 = op1 * op2
```

Варианты команды IMUL с двумя операндами:

```
IMUL reg, reg     IMUL reg, mem    IMUL reg, imm;
```

– три операнда:

```
IMUL op1, op2, op3
op1 = op2 * op3
```

Варианты команды IMUL с тремя операндами:

```
IMUL reg, reg, imm    IMUL reg, mem, imm    IMUL reg, imm, imm
```

Для деления чисел без знака предназначена команда **DIV**. У этой команды только один операнд – делитель, который должен находиться в регистре или в памяти. Местоположение делимого и результата задается неявно и зависит от размера операнда (табл. 3.2).

Таблица 3.2

Местоположение делимого и результата в команде DIV

Размер операнда	Делимое	Частное	Остаток
1 байт	*AX*	*AL*	*AH*
2 байта	*DX:AX*	*AX*	*DX*
4 байта	*EDX:EAX*	*EAX*	*EDX*

Пример:

```
div bl                ; AL = AX / BL
div bx                ; AX = DX:AX / BX
```

Команда **IDIV** используется для деления чисел со знаком, синтаксис ее такой же, как у команды **DIV**.

Арифметические команды изменяют флаги переполнения (*OF*), переноса (*CF*), знака (*SF*), нуля (*ZF*) и четности (*PF*) в соответствии со значением результата.

Пример линейной программы, осуществляющей консольный ввод данных, вычисление заданной функции и вывод результата.

```
.686
.model flat, stdcall
option casemap: none


; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\windows.inc
include C:\masm32\include\user32.inc
include C:\masm32\include\kernel32.inc
include C:\masm32\include\masm32.inc
includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\masm32.lib

; Сегмент данных
;-----------------------------------------
.data
B               sword   -6
D               sword   11
promptText      db      "Введите значение A:", 0Dh, 0Ah, 0
resultText      db      'Ответ: '
resStr          db      16 DUP(' '), 0
MsgExit         db      13,10,"Нажмите Enter для завершения", 0Ah, 0Dh,
0
MsgBoxCaption   db      'Оконный вывод', 0

.data?

X               SWORD   ?
A               SWORD   ?
buffer          db      10 DUP(?)
inbuf           db      100 DUP(?)
hOutput         dword   ?


; Сегмент кода
;-----------------------------------------
.code
start:
```

```
; Ввод данных
; вывод на экран приглашения ввести данные
      invoke StdOut, offset promptText

; ввод данных (строки) в буфер
      invoke StdIn, ADDR buffer, LENGTHOF buffer
; замена символа конца строки нулем
      invoke StripLF, ADDR buffer
; Преобразование в SDWORD
      invoke atol, ADDR buffer
; результат в EAX
      mov DWORD PTR A, EAX
; Вычисление Y = (A + B)*(B - 1)/(D + 8)
      mov CX, D
      add CX, 8                    ; CX = D + 8
      mov BX, B
      dec BX                       ; BX = B - 1
      mov AX, A
      add AX, D                    ; AX = A + D
      imul BX                      ; DX:AX = (A + D) * (B - 1)
      idiv CX                      ; AX = (DX:AX) : CX
      mov X, AX
; преобразование переменной X в строку
      invoke dwtoa, X, ADDR resStr
; вывод строки "Ответ: "
      invoke StdOut, ADDR resultText
; сброс EAX
      xor EAX, EAX
; вывод сообщения "Нажмите Enter..." с помощью функции WriteFile
; получаем дескриптор стандартного устройства вывода
      invoke GetStdHandle, STD_OUTPUT_HANDLE
; и сохраняем его в hOutput
      mov hOutput, eax
; ждем нажатия Enter
invoke WriteFile, hOutput, ADDR MsgExit, LENGTHOF MsgExit, NULL, NULL
      invoke StdIn, ADDR inbuf, LENGTHOF inbuf
; завершение процессов Windows
      invoke ExitProcess, NULL
; окончание сегмента start
end start
```

Лабораторная работа № 3

Цель работы:

– приобретение навыков использования функций ввода/вывода библиотеки *masm32.lib*, а также функций ввода/вывода API Windows;

– приобретение навыков программирования линейных алгоритмов;

– получение навыков работы с отладчиком.

Задания для лабораторной работы

1. Написать программу, вычисляющую заданное в соответствии с номером студента в журнале выражение (табл. 3.3).

Таблица 3.3

Номер	Функция	Ввод	Вывод
1	$y = (b^2 - (c + 1) \cdot d) / b$	***StdIn***	***WriteFile***
2	$y = a \cdot x / 2 - (a + b) / 2$	***ReadConsole***	***StdOut***
3	$y = (k - 5)^2 / 4 + 2 \cdot k$	***StdIn***	***WriteFile***
4	$y = a^3 / 3 - c \cdot (x + 3)$	***ReadConsole***	***MessageBox***
5	$y = a \cdot x - 3 \cdot (b + 3 / k)$	***StdIn***	***WriteConsole***
6	$y = 3 \cdot a \cdot x / [5 \cdot (b - 5)]$	***ReadConsole***	***StdOut***
7	$y = a \cdot (a + b / 4) / (k - 1)$	***StdIn***	***WriteFile***
8	$y = c \cdot (c + b / 4) / (k - 1)$	***ReadConsole***	***MessageBox***
9	$y = a \cdot j - j^2 / (k + 2)$	***StdIn***	***WriteConsole***
10	$y = a \cdot j - j^2 / (k + 2)$	***ReadConsole***	***StdOut***
11	$y = a / c - k + (d + 1) \cdot 5$	***StdIn***	***WriteFile***
12	$y = a \cdot b / 2 - k + a / 2 - b$	***ReadConsole***	***MessageBox***
13	$y = (b^2 - (c + 1) \cdot \mathrm{d}) / b$	***StdIn***	***WriteConsole***
14	$y = b \cdot (c - d) - c / (d - 1)$	***ReadConsole***	***StdOut***
15	$y = (a + b) / d - d^2 \cdot a - b$	***StdIn***	***WriteFile***
16	$y = (m - 5) \cdot (m + 2) + m + a / 2$	***ReadConsole***	***MessageBox***
17	$y = a^2 / 2 - b^3 / (4 - a) + b$	***StdIn***	***WriteConsole***
18	$y = (a - b^2) / (x - a) + a^2 - c$	***ReadConsole***	***StdOut***
19	$y = (1 - a)^2 / c + k - 1 + c / 2$	***StdIn***	***WriteFile***
20	$y = a \cdot c^2 - b \cdot a / c + a / b$	***ReadConsole***	***MessageBox***
21	$y = a \cdot x \cdot (b - a) / 4 + a^2 - 2$	***StdIn***	***WriteConsole***
22	$y = q^3 - 2 \cdot a \cdot q + a^2 / q$	***ReadConsole***	***StdOut***
23	$y = a / c - k + (d + 1) \cdot 5$	***StdIn***	***WriteFile***
24	$y = q^2 / 3 - a \cdot d + 5$	***ReadConsole***	***MessageBox***

Окончание табл. 3.3

Номер	Функция	Ввод	Вывод
25	$y = a \cdot x^2 - b \cdot x / a + x / (x + a)$	***StdIn***	***WriteConsole***
26	$y = b^2 \cdot (x + d) + (d - 1) / c$	***ReadConsole***	***StdOut***
27	$y = (t^3 - 1) / (j - 4) - 5$	***StdIn***	***WriteFile***
28	$y = (a^2 - b^2) / 2 + a \cdot (k + 1)$	***ReadConsole***	***MessageBox***
29	$y = (a - c)^2 + 2 \cdot a \cdot c / k$	***StdIn***	***WriteConsole***
30	$y = (b^2 - 2 \cdot b) / (3 \cdot a + b)$	***ReadConsole***	***StdOut***

ПРИМЕЧАНИЕ

Перед вычислением программа должна выполнить запрос значений параметров, обозначенных в выражениях буквенными символами, например, так:

```
Введите значение a:
56
Введите значение b:
-67
```

Затем программа должна произвести вычисление, и вывести его результат, например так:

```
Ответ: y = 347.
```

2. Открыть программу в отладчике. Выполнить программу в пошаговом режиме. После выполнения каждого шага заносить данные в табл. 3.4.

Таблица 3.4

Результат работы в отладчике*

Регистры				Флаги				Память
EAX	*EBX*	...	...	*FC*	*FZ*	…	…	

*В таблицу должны быть включены те элементы (регистры, флаги и пр.), которые имеют место в конкретном варианте программы.

3. По результатам п. 1 и 2 сделать выводы.

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Текст программ с комментариями.
5. Результаты выполнения программы в пошаговом режиме в отладчике.
6. Выводы.

Контрольные вопросы

1. Как сложить два операнда, если их размер превышает 32 бита?
2. Где находится произведение в результате умножения беззнаковых чисел?
3. Как выполнить деление беззнаковых чисел?
4. Что следует использовать для умножения знаковых чисел с непосредственным числом?
5. Какая команда позволяет устанавить бит в указанной позиции?
6. Какая команда сбрасывает бит в указанной позиции?
7. Какая команда позволяет проверить, установлен ли определенный бит в операнде?
8. Какая команда выполняет логическое сравнение двух операндов?
9. Какие функции используются для ввода и вывода информации на консоль в MASM32?
10. Как в MASM32 преобразовать текстовую строку в число и обратно?

Глава 4. Программирование арифметического сопроцессора

Арифметический сопроцессор (*FPU – Floating-Point Unit*) является частью архитектуры *x*86 и предназначен для выполнения операций с плавающей запятой. Он предоставляет аппаратную поддержку для высокоточных вычислений, включая операции сложения, вычитания, умножения, деления, сравнения и округления плавающих чисел.

Некоторые особенности арифметического сопроцессора:

– арифметический сопроцессор имеет восемь 80-битных регистров данных, называемых регистрами *FPU* (*ST*(0)…*ST*(7)). Эти регистры используются для хранения чисел с плавающей точкой и промежуточных результатов вычислений;

– сопроцессор поддерживает операции над числами с одинарной точностью (32 бита), двойной точностью (64 бита) и расширенной точностью (80 бит). Он предоставляет возможность работать с числами с плавающей точкой в форматах IEEE754;

– сопроцессор выполняет операции сложения, вычитания, умножения и деления чисел с плавающей точкой. Он также поддерживает операции сравнения для проверки отношений между числами;

– сопроцессор предоставляет возможность контролировать режимы округления для результатов операций с плавающей njxrjq. Режимы округления могут быть установлены для округления к ближайшему целому, округления вниз, округления вверх и округления к нулю;

– сопроцессор также содержит стек сопроцессора, который используется для хранения промежуточных результатов вычислений.

Команда **FINIT** инициализирует управляющие регистры сопроцессора определенными значениями. Данную команду используют перед первой командой сопроцессора в программе и в других случаях, когда необходимо привести сопроцессор в начальное состояние.

Команда **FLD** (*FLoaD*) используется для загрузки значения из памяти или регистра в верхний регистр *ST*(0).

При выполнении **FLD** значение из операнда-источника op2 (память или регистр) загружается в верхний регистр *ST*(0) стека сопроцессора. Все остальные значения в стеке сдвигаются вниз, так что значение, которое было в *ST*(0) перед выполнением **FLD**, перемещается в *ST*(1), значение из *ST*(1) перемещается в *ST*(2), и т. д.

В табл. 4.1 приведены некоторые команды передачи данных вещественного типа (4-, 8- или 10-байтный).

Таблица 4.1

Некоторые команды передачи данных

Команда	Операнд	Пояснение	Описание
FLD	op2	TOP_{SWR} – = 1; *ST*(0) = op2	Загрузка операнда в вершину *ST*(0) стека
FST	op1	op1 = *ST*(0)	Сохранение вершины *ST*(0) стека в память
FSTP	op1	op1 = *ST*(0); TOP_{SWR} + = 1	Сохранение вершины *ST*(0) стека в память с выталкиванием
FXCH	ST(i)	*ST*(0) ↔ *ST*(*i*)	Обмен значений *ST*(0) и *ST*(*i*)

Пример команды сопроцессора – сложение вещественных чисел:

```
FADD op1, op2
```

Команда складывает операнды `op1` и `op2` и помещает результат в `op1`.

Команда имеет следующие формы:

– `FADD op1` – источником является 32- или 64-битная переменная, содержащая вещественное число, а приемником *ST*(0);

– `FADD ST(0),ST(n); FADD ST(n),ST(0)` – источником и приемником являются регистры *FPU*;

– `FADD` – без операндов эквивалентна `FADD ST(0),ST(1).`

Некоторые арифметические команды вещественного типа приведены в табл. 4.2.

Таблица 4.2

Арифметические команды вещественного типа

Команда	Операнды	Пояснение	Описание
FADD	op1, op2	op1 = op1 + op2	Сложение вещественное
FSUB	op1, op2	op1 = op1 – op2	Вычитание вещественное
FMUL	op1, op2	op1 = op1 · op2	Умножение вещественное
FDIV	op1, op2	op1 = op1/op2	Деление вещественное

Для вывода результатов вычислений в консоль или окно необходимо двоичный код результата преобразовать в строку. Для этого можно использовать функцию **FpuFLtoA**.

Функция **FpuFLtoA** предназначена для вывода вещественного числа на экран. Данная функция не является API-функцией, а является внутренней MASM32-функцией и входит в специальную библиотеку *FPU.LIB*. Подробное описание функции можно найти в справке MASM32, которая обычно располагается в каталоге *\masm32\help\fpuhelp.chm*.

Функция **FpuFLtoA** может работать только с 80-битовыми числами и имеет четыре параметра:

```
FpuFLtoA (lpSrc1, lpSrc2, lpszDest, uID)
```

– `lpSrc1` – адрес 80-битового числа, которое будет выводиться на экран (этот параметр игнорируется, если *uID* имеет значение *SRC1_FPU*);

– lpSrc2 – беззнаковое целое, указывающее количество десятичных знаков после запятой или адрес беззнакового целого (в зависимости от параметра *uID*);

– lpszDest – адрес буфера, куда будут записаны символы, в которые преобразуется число. Буфер должен быть достаточно большим, чтобы вместить возвращаемые символы. Самое большое число может иметь 17 символов перед знаком десятичной точки, знак десятичной точки, 15 десятичных цифр после точки, и завершающий нуль (таким образом, максимум 34 символа);

– uID – флаги, управляющие работой функции. Один из флагов SRC1_? может объединяться с помощью операции **OR** только с одним из SRC2_? и одним из STR_? флагов (Флаг STR_REG не нужно объединять оператором **OR**, если строка должна быть возвращена в десятичном формате).

Значение uID флагов:

– SRC1_FPU – первый параметр функции Src не используется (уже в FPU);

– SRC1_REAL – первый параметр Src должен быть адресом 80-битного числа, хранящегося в обычной памяти;

– SRC2_DMEM – второй параметр Src2 должен быть адресом 32-битного беззнакового числа;

– SRC2_DIMM – второй параметр Src2 должен быть простым 32-битным беззнаковым числом;

– STR_REG – строка должна быть возвращена в десятичном формате;

– STR_SCI – строка должна быть возвращена в научном формате.

Для оконного вывода результатов необходимо изменить параметры линковщика в файле *Run.bat*, а именно:

```
C:\masm32\bin\link.exe /SUBSYSTEM:WINDOWS /LIBPATH:C:\masm32\lib\
Test.obj
```

Пример вычислений с использованием арифметического сопроцессора.

Вычислить 5 значений функции $Y_n = 4{,}3 \cdot (x^2 + 1)$ (x изменяется с шагом 1,7).

```
;-----------------------------------------
; Программа на masm32 для Windows
; Исследование работы арифметического сопроцессора
; Вычислить 5 значений функции Yn = 4,3*(x^2 + 1)
; (x изменяется с шагом 1,7)
;-----------------------------------------
```

```
.686                                ; в программе будут использоваться
                                    ; команды процессора Pentium Pro
.model flat, stdcall                ; модель памяти и соглашение
                                    ; о передаче параметров
option casemap :none                ; включается чувствительность
                                    ; к регистру

; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\windows.inc
include C:\masm32\include\kernel32.inc
include C:\masm32\include\user32.inc
include C:\masm32\include\fpu.inc
; содержит прототип функции FpuFLtoA
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\fpu.lib

; Сегмент даных
;-----------------------------------------
.data                               ; инициализированные данные
MsgBoxTitle byte "Операции в сопроцессоре x87", 0
                                    ; заголовок окна
MsgBoxText  db   "Вычисление функции Yn = 4,3*(x^2 + 1),", 13,
                 "где x изменяется с шагом 1,7", 13, 13,
                 "y1="
res1        db   14 DUP(0), 10, 13
                                    ; зарезервировать 14 байт для первого
                                    ; результата и поместить туда 0
            db   "y2="
res2        db   14 DUP(0), 10, 13
            db   "y3="
res3        db   14 DUP(0), 10, 13
            db   "y4="
res4        db   14 DUP(0), 10, 13
            db   "y5="
res5        db   14 DUP(0), 10, 13

CrLf  equ 0A0Dh
y1          TBYTE 0.0               ; тип 80 бит без знака (TBYTE = dt)
y2          dt    0.0
y3          dt    0.0
y4          dt    0.0
y5          dt    0.0
x           DWORD 3.0               ; тип 32 бита без знака (DWORD = dd)
```

```
op1         dd    4.3
op2         dd    1.0
zero        dd    0.0
step        dd    1.7

; Сегмент кода
;-----------------------------------------
.code

start:                          ; метка (точка входа в программу)
      finit                     ; инициализация регистров FPU
                                ; (CWR = 037Fh, SWR = 0h, TWR = FFFFh,
                                ; DPR = 0h, IPR = 0h)
      mov ecx, 5                ; счетчик X
m1:                             ; метка начала цикла
      fld x                     ; x^2
      fmul x
      fadd op2                  ; x^2+1
      fmul op1                  ; 4,3*(x^2+1)
      fld x                     ; увеличение X на величину шага
      fadd step
      fstp x
      loop m1                   ; если ecx = ecx - 1 ≠ 0, переходим на m1
      fstp y5                   ; сохраняем стек в память
      fstp y4
      fstp y3
      fstp y2
      fstp y1

; преобразование результатов вычислений в массив символов
invoke FpuFLtoA, addr y1, 10, addr res1, SRC1_REAL or SRC2_DIMM
      mov word ptr res1 + 14, CrLf
invoke FpuFLtoA, addr y2, 10, addr res2, SRC1_REAL or SRC2_DIMM
      mov word ptr res2 + 14, CrLf
invoke FpuFLtoA, addr y3, 10, addr res3, SRC1_REAL or SRC2_DIMM
      mov word ptr res3 + 14, CrLf
invoke FpuFLtoA, addr y4, 10, addr res4, SRC1_REAL or SRC2_DIMM
      mov word ptr res4 + 14, CrLf
invoke FpuFLtoA, addr y5, 10, addr res5, SRC1_REAL or SRC2_DIMM

; вывод результатов вычислений
invoke MessageBox, NULL, addr MsgBoxText, addr MsgBoxTitle,
MB_ICONINFORMATION
```

```
invoke ExitProcess, NULL     ; функция завершения с параметром NULL
end start                    ; окончание программы
```

Лабораторная работа № 4

Цель работы:

– знакомство с работой арифметического сопроцессора;
– исследование инструкций сопроцессора;
– получение навыков работы с отладчиком.

Задания для лабораторной работы

1. Согласно номеру студента в журнале группы выбрать вариант задания и написать программу вычисления заданного выражения на языке Ассемблера.

2. Выполнить программу в пошаговом режиме в отладчике и заполнить таблицу, отображающую состояние регистров и флагов сопроцессора.

3. Прокомментировать полученный результат.

Варианты заданий

Исследовать выполнение арифметических операций сопроцессором. Результат вывести на экран функцией MessageBox.

1. Вычислить 6 значений функции $Y_n = 25 \cdot x^3 - 2{,}1$ (x изменяется с шагом 0,2).

2. Вычислить 5 значений функции $Y_n = 7 \cdot x^3/(2 \cdot x^2 + 1{,}6)$ (x изменяется от 1 с шагом 4).

3. Вычислить 4 значения функции $Y_n = 37/(2 \cdot x^2 + 7{,}3)$ (x изменяется от 1 с шагом 2).

4. Вычислить 5 значений функции $Y_n = 5{,}1 \cdot x^2 + 5{,}3$ (x изменяется от 4,7 с шагом 3). Результат округлить к ближайшему целому числу и вывести.

5. Вычислить 6 значений функции $Y_n = 4 \cdot x/(x + 5)$ (x изменяется от 3 с шагом 1,25). Результат округлить к целому числу.

6. Вычислить 4 значения функции: $Y_n = 125/(3 \cdot x^2 - 1{,}1)$ (x изменяется от 3 с шагом 1,5). Результат округлить в меньшую сторону.

7. Вычислить 4 значения функции $Y_n = 2{,}5 \cdot x^2 - 3{,}2$ (x изменяется от 4 с шагом 1).

8. Вычислить 3 значения функции $Y_n = 25 \cdot x^2 + 2{,}1$ (x изменяется от 3 с шагом 2,5).

9. Вычислить 4 значения функции $Y_n = 150/(x^2 - 7)$ (x изменяется от 2 с шагом 3,1).

10. Вычислить 6 значений функции $Y_n = 256/(3 \cdot x^2 + 31)$ (x изменяется от 2 с шагом 3).

11. Найти первое значение аргумента функции $Y_n = 7 \cdot (x + 0{,}3)$, при котором младшие целые цифры результата выполнения функции будут равны 15 (x изменяется от 2 с шагом 3,5).

12. Найти целое значение аргумента, при котором функция $Y_n = 10/(x^3 + 1{,}7)$ станет меньше 0,3 (x изменяется от 2 с шагом 2,5).

13. Найти значение x, при котором функция $Y_n = 3 \cdot x/4 - 6$ будет равна 12,5.

14. Найти значение x, при котором выполняется равенство $8 \cdot \text{arctg } 0{,}1 + \text{arctg } x = \pi/4$.

15. Определить номер элемента функции $Y_n = 2 \cdot x + 5$, при котором сумма элементов превысит 12 000.

16. Найти значение x, при котором функция $\lg(2 \cdot x + 3)$ будет больше 2,5.

17. Найти целое значение аргумента, при котором функция $Y = 5{,}6 \cdot x/3 \cdot x^2$ будет больше 125.

18. Найти целое значение аргумента, при котором функция $Y = 2 \cdot x + 30$ будет больше 100.

19. Найти первое значение аргумента функции $Y = 9 \cdot (x^2 + 0{,}6)$, при котором младшие целые цифры результата выполнения функции будут равны 12 (x изменяется от 3 с шагом 5,5).

20. Найти целое значение аргумента, при котором функция $Y_n = 2 \cdot x/5 + 4$ превысит 400.

21. Вычислить 5 значений функции $Y_n = 5{,}2 \cdot \ln(\sin x)$ (x изменяется в градусах с шагом 1,5).

22. Вычислить 4 значения функции $Y_n = 5 \cdot x + \cos x$ (x изменяется от 0,4 с шагом 0,15).

23. Вычислить 6 значений функции $Y_n = 4{,}5 \cdot \lg(\text{tg } x)$ (x изменяется в градусах с шагом 10).

24. Вычислить 6 значений функции $Y_n = 12 \cdot x - \sin x$ (x изменяется с шагом 0,05).

25. Вычислить 8 значений функции $Y_n = 3{,}3 \cdot \log_2(x^2 + 1)$ (x изменяется с шагом 0,3).

26. Вычислить 4 значения функции $Y_n = 5{,}1(\sin x)$ (x изменяется в градусах с шагом 8,5).

27. Вычислить 6 значений функции $Y_n = \lg(x^2 + \sqrt{x})$ (x изменяется с шагом 0,3).

28. Вычислить 5 значений функции $Y_n = 4{,}3 \cdot (x^2 + 1)$ (x изменяется с шагом 1,7).

29. Вычислить 6 значений функции $Y_n = 6{,}2 \cdot \lg(\cos x)$ (x изменяется в градусах с шагом 1,5).

30. Вычислить 7 значений функции $Y_n = 3{,}1 + 2/\cos x$ (x изменяется в градусах от 10 с шагом 8).

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Текст программ с комментариями.
5. Результаты выполнения программы в пошаговом режиме в отладчике.
6. Выводы.

Контрольные вопросы

1. Какие задачи выполняет арифметический сопроцессор?
2. Какие операции над числами с плавающей точкой поддерживает сопроцессор?
3. Что такое стек сопроцессора и для чего он используется?
4. Какие форматы чисел с плавающей запятой поддерживаются в арифметическом сопроцессоре?
5. Каким образом происходит добавление и удаление значений со стека сопроцессора?
6. Что происходит при переполнении стека сопроцессора?
7. Какими способами сопроцессор взаимодействует с основным процессором?

Глава 5. Программирование разветвляющихся алгоритмов

Команды сравнения данных

Команда **CMP** вычитает операнд op2 из операнда op1 и, в зависимости от полученного результата, устанавливает флаги состояния процессора. При этом, в отличие от команды SUB, значение операнда op1 не изменяется:

```
CMP op1, op2
```

В команде **CMP** используются аналогичные команде AND типы операндов.

Команда **CMP** может изменять состояние следующих флагов:

– *CF* (флаг переноса);

– *ZF* (флаг нуля);

– *SF* (флаг знака);

– *OF* (флаг переполнения);

– *AF* (флаг служебного переноса);

– *PF* (флаг четности).

Они устанавливаются в зависимости от значения, которое было бы получено в результате применения команды **SUB**. Например, как показано в табл. 5.1, после выполнения команды **CMP**, по состоянию флагов нуля (*ZF*) и переноса (*CF*) можно судить о величинах сравниваемых между собой беззнаковых операндов.

Таблица 5.1

Состояние флагов после сравнения беззнаковых операндов с помощью команды *CMP*

Отношение операндов	*ZF*	*CF*
op1 < op2	0	1
op1 = op2	1	0
op1 > op2	0	0

Если сравниваются два операнда со знаком, то, кроме флагов *ZF* и *CF*, нужно учитывать еще и флаг знака (*SF*), как показано в табл. 5.2.

Таблица 5.2

Состояние флагов после сравнения операндов со знаком с помощью команды *CMP*

Отношение операндов	Состояние флагов
op1 < op2	$SF \neq OF$ и $ZF = 0$
op1 = op2	$ZF = 1$
op1 > op2	$SF = OF$ и $ZF = 0$

Команда **CMP** очень важна, поскольку она используется практически во всех основных условных логических конструкциях. Если после команды **CMP** поместить команду условного перехода, то полученная конструкция на языке ассемблера будет аналогична оператору **if** языка высокого уровня.

Пример. При сравнении числа 5, находящегося в регистре *EAX*, с числом 10, устанавливается флаг переноса *CF*, поскольку при вычитании числа 10 из 5 происходит заем единицы:

```
MOV eax, 5
CMP eax, 10                ; CF = 1
```

При сравнении содержимого регистров *EAX* и *ECX*, в которых содержатся одинаковые числа 15, устанавливается флаг нуля (*ZF*), так как в результате вычитания этих чисел получается нулевое значение:

```
MOV eax, 15
MOV ecx, 15
CMP eax, ecx               ; ZF = 1
```

С помощью команды **OR** можно определить, какое значение находится в регистре (отрицательное, положительное или нуль). Для этого вначале нужно выполнить команду **OR**, указав в качестве операндов один и тот же регистр, например:

```
OR dl, dl
```

а затем – проанализировать значения флагов, как показано в табл. 5.3.

Таблица 5.3

Определение значения числа по флагам состояния процессора

ZF	*SF*	**Значение**
0	0	Больше нуля
1	0	Равно нулю
0	0	Меньше нуля

Команда **TEST** выполняет операцию поразрядного логического **И** между соответствующими парами битов операндов и, в зависимости от полученного результата, устанавливает флаги состояния процессора. При этом, в отличие от команды **AND**, значение операнда_1 не изменяется. В команде **TEST** используются аналогичные команде **AND** типы операндов. Обычно команда **TEST** применяется для анализа значения отдельных битов числа по маске.

Пример. С помощью команды **TEST** определить состояние сразу нескольких битов числа. Предположим, мы хотим узнать, установлен ли нулевой и третий биты регистра *AL*. Для этого можно воспользоваться такой командой:

```
TEST al, 00001001b     ; проверяем биты 0 и 3
```

Команда TEST всегда сбрасывает флаги переполнения (*OF*) и переноса (*CF*). Кроме того, она устанавливает значения флагов знака (*SF*), нуля (*ZF*) и четности (*PF*) в соответствии со значением результата выполнения операции логического И (как и команда AND).

Установка и сброс отдельных флагов состояния процессора

Для установки или сброса флагов нуля (*ZF*), знака (*SF*), переноса (*CF*) и переполнения (*OF*) существуют несколько способов, и большинство из них изменяет значение операнда получателя данных. Чтобы установить флаг нуля (*ZF*), необходимо выполнить команду **AND** с нулевым операндом, а для сброса этого флага – команду **OR** с единичным операндом:

```
AND al, 0                  ; ZF = 1
OR al, 1                   ; ZF = 0
```

Для установки флага знака (*SF*) выполните команду **OR**, у которой старший бит операнда установлен в 1, а для сброса этого флага – команду **AND**, у которой старший бит операнда сброшен в 0, как показано ниже:

```
AND al, 7Fh                ; SF = 0
OR al, 80h                 ; SF = 1
```

Для установки и сброса флага переноса (*CF*) предусмотрены специальные команды: **STC** (*SeT Carry flag*) и **CLC** (*CLear Carry flag*):

```
CLC                        ; CF = 0
STC                        ; CF = 1
```

Чтобы установить флаг переполнения (*OF*), нужно сложить два положительных числа так, чтобы в результате получилась отрицательная сумма. Для сброса этого флага достаточно выполнить команду **OR** с нулевым операндом, как показано ниже:

```
MOV al, 7Fh                ; al = +127
INC al                     ; al = 80h (-128),  OF = 1
OR al, 0                   ; OF = 0
```

Команды безусловного перехода

Команда безусловного перехода имеет следующий синтаксис:

```
JMP <операнд>
```

Операнд указывает адрес перехода. Существует два способа указания этого адреса, соответственно различают прямой и косвенный переходы.

Прямой переход. Если в команде перехода указывается метка команды, на которую надо перейти, то переход называется прямым:

```
JMP m1
…
m1:
MOV eax, x
```

Косвенный переход. При косвенном переходе в команде перехода указывается не адрес перехода, а регистр или ячейка памяти, где этот адрес находится. Содержимое указанного регистра или ячейки памяти рассматривается как абсолютный адрес перехода. Косвенные переходы используются в тех случаях, когда адрес перехода становится известен только во время работы программы:

```
JMP reg32/mem32
```

Команды условного перехода

На языке Ассемблера с помощью набора команд сравнения и условного перехода можно реализовать логическую структуру любой сложности. В языке высокого уровня любой условный оператор выполняется в два этапа. Сначала вычисляется значение условного выражения, а затем, в зависимости от его результата, выполняются те или иные действия. Проводя аналогию с языком Ассемблера, можно отметить, что сначала выполняются такие команды, как **CMP**, **AND** или **SUB**, влияющие на флаги состояния процессора. Затем выполняется команда условного перехода, которая анализирует значение нужных флагов и, в случае если они установлены, выполняет переход по указанному адресу.

Пример. С помощью команды **CMP** значение регистра *AL* сравнивается с нулем. Команда **JZ** (*Jump if Zero*, или переход, если ноль) передает управление по метке **m1**, если в результате выполнения команды **CMP** был установлен флаг *ZF*:

```
CMP al, 0
JZ  m1                          ; переход если ZF=1
…
m1:
```

Команда условного перехода передает управление по указанной метке в случае, если установлен соответствующий флаг состояния процессора. Если флаг сброшен, то выполняется следующая за ней команда. Синтаксис команд условного перехода следующий:

```
Jxx <операнд>
```

Здесь вместо параметра XX нужно подставить аббревиатуру нужного условия, которая будет определять состояние одного или нескольких флагов состояния процессора.

При выполнении любой команды условного перехода процессор вначале проверяет состояние соответствующих флагов регистра EFLAGS. Если он обнаруживает, что флаги, соответствующие данному условию, установлены, выполняется переход по указанной метке. В противном случае управление передается команде, следующей за командой условного перехода.

В системе команд предусмотрено много типов команд условного перехода. Удобно разбить весь этот набор команд на четыре группы:

– выполняющие переход в зависимости от значения флагов состояния процессора;

– выполняющие переход в зависимости от равенства операндов или равенства нулю регистра *ECX* (*CX*);

– использующиеся после команд сравнения беззнаковых операндов;

– использующиеся после команд сравнения операндов со знаком.

В табл. 5.4 перечислены команды первой группы, выполняющие переход в зависимости от значения флагов состояния процессора: *ZF*, *CF*, *OF*, *PF* и *SF*.

Таблица 5.4

Команды, выполняющие переход, в зависимости от значения флагов состояния процессора

Мнемокод	Описание	Состояние флага
JZ	Переход, если 0	*ZF* = 1
JNZ	Переход, если не 0	*ZF* = 0
JC	Переход, если перенос	*CF* = 1
JNC	Переход, если нет переноса	*CF* = 0
JO	Переход, если переполнение	*OF* = 1
JNO	Переход, если нет переполнения	*OF* = 0
JS	Переход, если знак установлен	*SF* = 1
JNS	Переход, если знак не установлен	*SF* = 0
JP	Переход, если флаг четности установлен	*PF* = 1
JNP	Переход, если флаг четности не установлен	*PF* = 0

Сравнение на равенство. В табл. 5.5 перечислены команды, выполняющие переход в зависимости от равенства операндов или равенства нулю регистра *ECX* (*CX*):

```
CMP op1, op2
```

Таблица 5.5

Команды, выполняющие переход, в зависимости от равенства операндов

Мнемокод	Описание	Состояние флага
JE	Переход, если op1 = op2	*ZF* = 1
JNE	Переход, если op1 ≠ op2	*ZF* = 0
JCXZ	Переход, если регистр *CX* = 0	–
JECXZ	Переход, если регистр *ECX* = 0	–

Сравнение беззнаковых чисел. Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов, перечислены в табл. 5.6. Этот тип команд условного перехода используется в случае, если нужно сравнить два беззнаковых числа, таких как 7Fh и 80h, и при этом предполагается, что первое число (7Fh) должно быть меньше второго (80h), а не наоборот, как в случае операндов со знаком.

Таблица 5.6

Команды условного перехода, использующиеся после команд сравнения беззнаковых операндов

Мнемокод	Описание	Состояние флага
JA	Переход, если op1 > op2	*CF* = 0 и *ZF* = 0
JNBE		
JAE	Переход, если op1 ≥ op2	*CF* = 0 или *ZF* = 1
JNB		
JB	Переход, если op1 < op2	*CF* = 1 и *ZF* = 0
JNAE		
JBE	Переход, если op1 ≤ op2	*CF* = 1 или *ZF* = 1
JNA		

Сравнение чисел со знаком. Команды условного перехода, использующиеся после команд сравнения операндов со знаком, перечислены в табл. 5.7. Этот тип команд условного перехода используется в случае, если сравниваемые операнды должны интерпретироваться как числа со знаком. Например, при сравнении чисел 7Fh и 80h результат будет зависеть от того, являются ли эти числа беззнаковыми или содержат знак. Поэтому команды **JA** и **JG** будут работать по-разному:

```
MOV al, 7Fh                ; 7Fh = +127
CMP al, 80h                ; 80h может быть +128 или -128
JA  IsAbove                ; нет перехода ввиду +127  < +128
JG  IsGreater              ; переход ввиду +127 > -128
```

В этом примере команда **JA** не выполняет переход, так как беззнаковое число 7Fh меньше, чем беззнаковое число 80h. В то же время команда **JG** выполняет переход, поскольку число +127 больше, чем –128.

Таблица 5.7

Команды условного перехода, использующиеся после команд сравнения операндов со знаком

Мнемокод	Описание	Состояние флага
JG	Переход, если op1 > op2	*SF* = *OF* и *ZF* = 0
JNLE		
JGE	Переход, если op1 ≥ op2	*SF* = 0 или *ZF* = 1
JNL		

Окончание табл. 5.7

Мнемокод	Описание	Состояние флага
JL	Переход, если op1 < op2	$SF \neq OF$ и $ZF = 0$
JNGE		
JLE	Переход, если op1 ≤ op2	$SF \neq OF$ или $ZF = 1$
JNG		

Пример линейной программы, осуществляющей консольный ввод данных, вычисление заданной функции и вывод результата:

$$y = \begin{cases} 5 \cdot x, \text{ если } x < 10 \\ \dfrac{100}{x}, \text{ если } x = 10 \\ x - 10, \text{ в остальных случаях} \end{cases}$$

```
.686
.model flat, stdcall
option casemap: none


; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\windows.inc

include C:\masm32\include\user32.inc

include C:\masm32\include\kernel32.inc
include C:\masm32\include\masm32.inc
includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\masm32.lib

; Сегмент данных
;-----------------------------------------
.data

sConsTitl  BYTE  "Лабораторная работа №1", 0
strPrompt  BYTE  "Введите значение X:", 0Dh, 0Ah, 0
MsgExit    BYTE  0Dh, 0Ah, "Для завершения нажмите Enter...", 0Dh, 0Ah, 0
resultText BYTE  "Ответ: "
resultStr  BYTE  16 DUP(' '), 0
```

```
.data?

hStdout     DWORD ?
hStdin      DWORD ?
cWritten    DWORD ?
buffer      BYTE  20 DUP(?)   ; буфер ввода/вывода
x           WORD  ?           ; переменная

; Сегмент кода
;-----------------------------------------
.code

start:

; Ввод данных
; вывод заголовка консоли
      invoke SetConsoleTitle, ADDR sConsoleTitle
; получение дескриптора консольного ввода
      invoke GetStdHandle, STD_INPUT_HANDLE
      mov hStdin, eax
; получение дескриптора консольного вывода
      invoke GetStdHandle, STD_OUTPUT_HANDLE
      mov hStdout, eax
; вывод сообщения в консоль
      invoke WriteConsole, hStdout, ADDR strPrompt, SIZEOF strPrompt,
ADDR cWritten
; чтение данных из консоли
      invoke ReadConsole, hStdin, ADDR buffer, SIZEOF buffer, ADDR
cWritten, NULL
; замена символа конца строки нулем
      invoke StripLF, ADDR buffer
; Преобразование в SDWORD
      invoke atol, ADDR buffer
      mov x, ax                  ; x = AX

;-----------------------------------------
; Вычисление функции
; y = 100/x, если x = 10
; y = 5*x, если x < 10
; y = x - 10, в остальных случаях

; Примечание! Результат вычислений помещается в EAX для удобства вывода
;-----------------------------------------
; Проверка значения переменной
      mov eax, 0                 ; EAX = 0
```

```
      mov ax, x                    ; AX = X
      cmp ax, 10                   ; сравниваем X и 10
      je m1                        ; переходим на метку "m1", если X = 10
      jl m2                        ; переходим на метку "m2", если X < 10
;-----------------------------------------
; Вычисление y = x - 10
      sub ax, 10                   ; AX = AX -10
      cwde                         ; расширяем AX до EAX с учетом знака
      jmp exit                     ; безусловный переход на метку "exit"
;-----------------------------------------
; Вычисление y = 100/x
m1:
      mov dx, 0                    ; DX = 0
      mov ax, 100                  ; AX = 100
      div x                        ; AX = DX:AX / X
      cwde                         ; расширяем AX до EAX с учетом знака
      jmp exit                     ; безусловный переход на метку "exit"
;-----------------------------------------
; Вычисление y = 5*x
m2:
      cwde                         ; расширяем AX до EAX с учетом знака
      imul eax, eax, 5             ; EAX = 5*X
      jmp exit                     ; безусловный переход на метку "exit"

;-----------------------------------------
; Вывод результата
exit:
; преобразование переменной EAX в строку
      invoke dwtoa, eax, ADDR resultStr

; вывод строки "Ответ: "
      invoke StdOut, ADDR resultText

; сброс EAX
      xor EAX, EAX
; вывод сообщения "Нажмите Enter..." с помощью функции WriteFile
            invoke WriteFile, hStdout, ADDR MsgExit, LENGTHOF MsgExit,
NULL, NULL
; ждем нажатия Enter
      invoke StdIn, ADDR buffer, LENGTHOF buffer
; завершение процессов Windows

      invoke ExitProcess, NULL
; окончание сегмента start
end start
```

Лабораторная работа № 5

Цель работы:

– приобретение навыков программирования разветвляющихся алгоритмов;

– получение навыков работы с отладчиком.

Задания для лабораторной работы

Разработать программу для вычисления функции $y = f(x)$ для аргумета $x \in [-16000; 16000]$. Подобрать тестовые значения аргуметна для проверки правильности работы программы. Вычислить значения функции y для тестовых значений x. Индивидуальные задания приведены в табл. 5.8.

Таблица 5.8

Варианты заданий к лабораторной работе № 5

Номер	Функция	Номер	Функция
1	$y = \begin{cases} 6 \cdot x, \text{если } x > 5 \\ x, \text{если } 1 \le x \le 5 \\ 4 \cdot x^2, \text{в остальных случаях} \end{cases}$	16	$y = \begin{cases} 3 \cdot x^2, \text{если } x > 10 \\ 3 \cdot x, \text{ если } x \le 2 \\ 4 \cdot x, \text{в остальных случаях} \end{cases}$
2	$y = \begin{cases} x^2, \text{если } x > 20 \\ \dfrac{100}{x}, \text{если } 1 \le x \le 20 \\ x + 2, \text{в остальных случаях} \end{cases}$	17	$y = \begin{cases} 2 \cdot x, \text{если } x > 10 \\ 4 \cdot x, \text{если } 1 \le x \le 10 \\ 6 \cdot x, \text{в остальных случаях} \end{cases}$
3	$y = \begin{cases} \dfrac{75}{x}, \text{если } x < -1 \\ 7 \cdot x, \text{если } -1 \le x \le 6 \\ \dfrac{x}{2}, \text{в остальных случаях} \end{cases}$	18	$y = \begin{cases} 5 \cdot x, \text{если } x > 2 \\ \lvert x \rvert, \text{если } -10 \le x \le -3 \\ 2 \cdot x^2, \text{в остальных случаях} \end{cases}$
4	$y = \begin{cases} 4 \cdot x, \text{если } x > 12 \\ x + 60, \text{если } 5 \le x \le 12 \\ 2 \cdot x^4, \text{в остальных случаях} \end{cases}$	19	$y = \begin{cases} 7 \cdot x, \text{если } x > 3 \\ \lvert x - 9 \rvert, \text{если } x \le 0 \\ x^2 + 2, \text{в остальных случаях} \end{cases}$
5	$y = \begin{cases} x - 90, \text{если } x > 10 \\ 6 \cdot x, \text{если } 1 \le x \le 9 \\ x^2, \text{в остальных случаях} \end{cases}$	20	$y = \begin{cases} x^3, \text{если } x > 8 \\ 2 \cdot x^2, \text{если } x \le 0 \\ x + 3, \text{в остальных случаях} \end{cases}$

Продолжение табл. 5.8

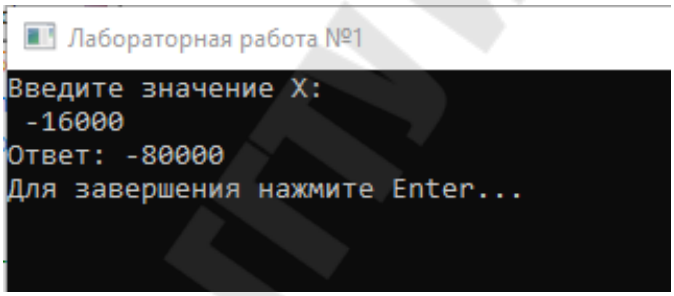
Номер	Функция	Номер	Функция
6	$y=\begin{cases}8\cdot x, \text{если } x>5\\ x^2, \text{ если } -5\le x\le 5\\ \lvert x+7\rvert, \text{в остальных случаях}\end{cases}$	21	$y=\begin{cases}x+99, \text{если } x>8\\ \frac{50}{x}, \text{если } 1\le x\le 8\\ x^2, \text{в остальных случаях}\end{cases}$
7	$y=\begin{cases}1-3\cdot x, \text{если } x>0\\ x-10, \text{если } -5\le x\le 0\\ x^2, \text{в остальных случаях}\end{cases}$	22	$y=\begin{cases}x^4, \text{если } x<10\\ x-20, \text{если } x>15\\ 5\cdot x, \text{в остальных случаях}\end{cases}$
8	$y=\begin{cases}x-9, \text{если } x>5\\ x^2, \text{если } 1\le x\le 5\\ x^3, \text{в остальных случаях}\end{cases}$	23	$y=\begin{cases}2\cdot x^2, \text{если } x>8\\ \lvert x-10\rvert, \text{если } x\le 5\\ x+7, \text{в остальных случаях}\end{cases}$
9	$y=\begin{cases}x-15, \text{если } x>10\\ \frac{100}{x}, \text{если } 4\le x\le 10\\ x-10, \text{в остальных случаях}\end{cases}$	24	$y=\begin{cases}6\cdot x, \text{если } x<10\\ x-5, \text{если } 10\le x\le 20\\ \lvert x+5\rvert, \text{в остальных случаях}\end{cases}$
10	$y=\begin{cases}\frac{88}{x}, \text{если } 1\le x\le 6\\ \lvert x-100\rvert, \text{если } x>6\\ \frac{x}{5}, \text{в остальных случаях}\end{cases}$	25	$y=\begin{cases}x^4, \text{если } x>4\\ 2\cdot x^3, \text{если } x\le 0\\ \lvert x-50\rvert, \text{в остальных случаях}\end{cases}$
11	$y=\begin{cases}8\cdot x, \text{если } x>2\\ \lvert x+1\rvert, \text{если } -10\le x\le -3\\ x^3, \text{в остальных случаях}\end{cases}$	26	$y=\begin{cases}x-77, \text{если } x>3\\ \lvert x-4\rvert, \text{если } x\le 0\\ x^2+2, \text{в остальных случаях}\end{cases}$
12	$y=\begin{cases}7\cdot x, \text{если } x>10\\ \lvert x-68\rvert, \text{если } 5\le x\le 10\\ 8-x^3, \text{в остальных случаях}\end{cases}$	27	$y=\begin{cases}x^3, \text{если } x>3\\ 9\cdot x^2, \text{если } x\le 0\\ x-6, \text{в остальных случаях}\end{cases}$
13	$y=\begin{cases}8\cdot x, \text{если } x>3\\ x^2, \text{если } 0\le x\le 3\\ \lvert x-7\rvert, \text{в остальных случаях}\end{cases}$	28	$y=\begin{cases}2\cdot x, \text{если } x>30\\ \frac{150}{x}, \text{если } 1\le x\le 15\\ x^2, \text{в остальных случаях}\end{cases}$

Окончание табл. 5.8

Номер	Функция	Номер	Функция
14	$y=\begin{cases}\lvert 1-3x\rvert, \text{если } x>0\\ 3\cdot x-7, \text{если } -50\le x\le 0\\ x^2, \text{в остальных случаях}\end{cases}$	29	$y=\begin{cases}4\cdot x^2, \text{если } x<10\\ x-20, \text{если } x>15\\ \lvert 5-x\rvert, \text{в остальных случаях}\end{cases}$
15	$y=\begin{cases}3\cdot x, \text{если } x>5\\ \frac{130}{x}, \text{если } -10\le x\le -1\\ \lvert x+5\rvert, \text{в остальных случаях}\end{cases}$	30	$y=\begin{cases}3\cdot x, \text{если } x>10\\ \lvert x-55\rvert, \text{если } 1\le x\le 10\\ 2\cdot x-9, \text{в остальных случаях}\end{cases}$

ПРИМЕЧАНИЕ

Перед вычислением программа должна выполнить запрос значения переменной *x*. Затем программа должна произвести вычисление и вывести его результат, например так:



2. Открыть исполняемый файл программы в отладчике. Выполнить программу в пошаговом режиме. После выполнения каждого шага заносить данные в табл. 5.9.

Таблица 5.9

Результаты выполнения программы в отладчике*

Регистры				Флаги				Память
EAX	*EBX*	…	…	*FC*	*FZ*	…	…	

*В таблицу должны быть включены те элементы (регистры, флаги и пр.), которые имеют место в конкретном варианте программы.

3. По результатам п. 1 и 2 сделать выводы.

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Текст программ с комментариями.
5. Блок-схема алгоритма.
6. Результаты выполнения программы в пошаговом режиме в отладчике.
7. Выводы.

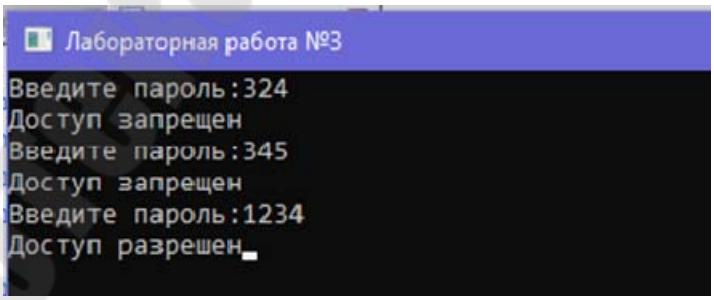
Рассмотрим еще один пример использования команд сравнения и перехода.

Напишите программу на Ассемблере Masm32, которая запрашивает у пользователя пароль и проверяет его на правильность. Если пароль введен правильно, программа выводит сообщение «*Доступ разрешен*». Если пароль введен неправильно, программа выводит сообщение «*Доступ запрещен*» и предлагает ввести пароль еще раз. Если пользователь вводит неправильный пароль более трех раз, программа завершает свою работу:

```
.686
.model flat, stdcall
option casemap :none

include C:\masm32\include\windows.inc
include C:\masm32\include\user32.inc
include C:\masm32\include\kernel32.inc
include C:\masm32\include\masm32.inc

includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\masm32.lib

.data
prompt              db    "Введите пароль:", 0
success             db    "Доступ разрешен", 0
failure             db     "Доступ запрещен", 0Dh, 0Ah, 0
max_attempts        db    3
current_attempt     db    0
password            db    "1234", 0
buffer              db    256 dup(?)
```

```
.code
start:
      mov current_attempt, 0       ; счетчик попыток инициализировать 0
ask_password:
      mov al, current_attempt      ; если превышено количество попыток,
                                   ; то выходим из программы
      cmp al, max_attempts
      jge exit_program
      invoke StdOut, addr prompt   ; выводим приглашение на ввод пароля
      invoke StdIn, addr buffer, 256
                                   ; считываем введенный пароль
      mov al, password             ; сравниваем введенный пароль и
                                   ; правильный пароль
      cmp al, buffer
      jne invalid_password         ; если пароль неправильный,
                                   ; переходим к invalid_password
      invoke StdOut, addr success  ; выводим сообщение об успешном входе
      invoke Sleep, 1000           ; пауза на 1000 мс
      jmp exit_program             ; завершаем программу
invalid_password:
      invoke StdOut, addr failure  ; выводим сообщение о неправильном
                                   ; вводе пароля
      inc current_attempt          ; увеличиваем счетчик попыток
      jmp ask_password             ; запрашиваем пароль еще раз
exit_program:
      invoke ExitProcess, 0        ; завершаем программу
      end start
```

Результат выполнения программы:



Лабораторная работа № 5а

Цель работы:

– приобретение навыков программирования разветвляющихся алгоритмов;

– получение навыков работы с отладчиком.

Задания для лабораторной работы

1. Составить блок-схему алгоритма и написать программу для решения задачи согласно индивидуальному заданию (табл. 5.10).

Таблица 5.10

Варианты заданий к лабораторной работе № 5а

Вариант	Задание
1	Дано натуральное число $n < 100$. Вывести на экран фразу «Мне n лет», учитывая, что при некоторых значениях n слово «лет» надо заменить на слово «год» или «годы»
2	Дано число m ($1 \le m \le 12$). Определить, сколько дней в месяце с номером m
3	С клавиатуры вводится натуральное число n. В зависимости от значения остатка r при делении числа n на 7 вывести на экран число n в виде $n = 7 \cdot k + r$
4	С клавиатуры вводится цифра m (от 1 до 4). Вывести на экран названия месяцев, соответствующих времени года с номером m (считать зиму временем года с номером 1)
5	В понедельник фирма работает с 9:00 до 16:00; во вторник, среду, четверг, пятницу – с 8:00 до 19:00; в субботу – с 10:00 до 15:00; воскресенье – выходной. По заданному номеру дня недели определить часы работы
6	Дано целое число C такое, что $\|C\| < 9$. Вывести это число в словесной форме, учитывая его знак
7	С клавиатуры вводится цифра m (от 1 до 12). Вывести на экран название месяца, соответствующего цифре
8	Дано число m ($1 \le m \le 12$). Определить полугодие, на которое приходится месяц с номером m и количество дней в том месяце (год не високосный)
9	Вводится число экзаменов $N \le 6$. Напечатать фразу «Мы успешно сдали N экзаменов», согласовав слово «экзамен» с числом N
10	Вводится число карандашей $N \le 10$. Вывести фразу «Я купил N карандашей», согласовав слово «карандаш» с числом N
11	Компания по снабжению электроэнергией взимает плату с клиентов по тарифу: 7 р. за 1 кВт/ч за первые 250 кВт/ч; 17 р. за кВт/ч, если потребление свыше 250, но не превышает 300 кВт/ч; 20 р. за кВт/ч, если потребление свыше 300 кВт/ч. Потребитель израсходовал n кВт/ч. Подсчитать плату

Продолжение табл. 5.10

Вариант	Задание
12	Студенты убирают урожай помидоров. При сборе до 50 кг в день работа оплачивается из расчета 30 коп. за 1 кг; при сборе от 50 до 75 кг в день – 50 коп. за 1 кг; при сборе от 75 до 90 кг в день – 65 коп. за 1 кг; при сборе свыше 90 кг в день – 70 коп. за 1 кг плюс 20 р. премия. Студент собрал *n* кг за день. Определить его заработок
13	В японском календаре был принят 60-летний цикл, состоящий из пяти 12-летних подциклов. Внутри подцикла годы носили названия животных: мыши, коровы, тигра, зайца, дракона, змеи, лошади, овцы, обезьяны, курицы, собаки и свиньи. Попарно годы в подцикле обозначались названиями цвета: зеленый, красный, желтый, белый и черный. По номеру года определить его название по японскому календарю, считая за начало очередного цикла 1984 г. – год зеленой мыши (1985 – год зеленой коровы, 1986 – год красного тигра, 1987 – год красного зайца и т. д.)
14	Из трех чисел *a*, *b* и *c* выбрать те, модули которых не меньше 4
15	Напечатать три заданных числа *a*, *b* и *c* сначала в порядке их возрастания, затем – в порядке убывания
16	Школьники сдают нормы по прыжкам в длину. Если длина прыжка больше 250 см, то оценка – 5, если от 200 см до 250 – оценка 4; от 150 см до 200 см – оценка 3; если меньше 150 см – 2. Выставить школьнику оценку, если известна длина его прыжка
17	Вывести на экран произведение трех заданных чисел, если сумма этих чисел больше 10, иначе вывести наименьшее из этих чисел
18	Определить, есть ли среди трех заданных чисел четные
19	Составить программу, выясняющую делится ли натуральное число *n* нацело на натуральное число *m*
20	Составить программу нахождения из трех чисел наибольшего и наименьшего
21	Дано 5 целых чисел. Составить программу, выбирающую из них те, которые принадлежат интервалу (6, 20)
22	Определить, есть ли среди трех заданных чисел нечетные
23	Попадет ли точка $A(a1, a2)$ в квадрат с заданной длиной полустороны *n* с центром в начале координат?
24	Какая из точек $A(a1, a2)$ или $B(b1, b2)$ находится ближе к началу координат?
25	Из трех чисел *a*, *b* и *c* выбрать те, модули которых больше 5
26	Дано 5 целых чисел. Составить программу, выбирающую из них те, которые не принадлежат интервалу (–5, +5)

Окончание табл. 5.10

Вариант	Задание
27	Попадет ли точка $A(a1, a2)$ в прямоугольник с заданными длинами полусторон n и m с центром в начале координат?
28	Вывести на экран сумму трех заданных чисел, если произведение этих чисел больше 10, иначе вывести наибольшее из этих чисел
29	Какая из точек $A(a1, a2)$ или $B(b1, b2)$ находится дальше от начала координат?
30	Дано число m ($1 \le m \le 7$). Вывести на экран название дня недели, который соответствует этому номеру. Значение 1 соответствует понедельнику

2. Выполнить тесты разработанной программы.

3. Открыть исполняемый файл программы в отладчике. Выполнить программу в пошаговом режиме. После выполнения каждого шага заносить данные в табл. 5.11.

Таблица 5.11

Результаты выполнения программы в отладчике*

Регистры				Флаги				Память
EAX	*EBX*	…	…	*FC*	*FZ*	…	…	

*В таблицу должны быть включены те элементы (регистры, флаги и пр.), которые изменяются в конкретном варианте программы.

4. По результатам п. 1–3 сделать выводы.

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Блок-схема алгоритма.
5. Текст программ с комментариями.
6. Скриншоты экрана с тестами.
7. Результаты выполнения программы в пошаговом режиме в отладчике.
8. Выводы.

Контрольные вопросы

1. Какие команды используются для сравнения значений в Ассемблере?

2. Какая команда выполняет условный переход, если два значения равны?

3. Какая команда выполняет условный переход, если значение больше другого значения?

4. Какая команда выполняет условный переход, если значение меньше или равно другому значению?

5. Какая команда выполняет безусловный переход к определенной метке (ярлыку) в программе?

6. Какие флаги устанавливаются после выполнения команды сравнения?

7. Какие команды используются для безусловного перехода на определенную метку в программе?

8. Какие команды могут быть использованы для создания условия перехода на основе флагов?

Глава 6. Обработка массивов данных

Работа с массивами на языке Ассемблера x86 в нотации MASM32 для Windows включает в себя использование различных инструкций для доступа к элементам массива, обхода массива, а также выполнения различных операций над элементами массива.

Для начала работы с массивом необходимо определить его размер и выделить под него память. Пример определения массива размером 10 элементов:

```
.data
array       DWORD       10 DUP(0)
```

В данном примере создается массив array размером 10 элементов, все элементы которого равны 0. Используется директива *DUP*(0), которая указывает, что каждый элемент массива должен быть инициализирован значением 0.

Или так:

```
A     BYTE    20 DUP(30 DUP(?))
```

При этом резервируется 600 байт, т. е. матрица *A* размером 20 x 30, элементы которой расположены в памяти следующим обра-

зом: первые 30 байтов – элементы первой строки матрицы, следующие 30 байтов – элементы второй строки и т. д.

Для доступа к элементам массива используются индексы. Индексация массива начинается с 0. Для доступа к элементу массива по его индексу используется инструкция *MOV* с использованием регистра индекса (например, *EAX*):

```
MOV EAX, array[0]       ; загрузка первого элемента массива в регистр EAX
```

Для обхода массива используются **циклы**. Например, для обхода массива и увеличения каждого элемента на 1 можно использовать следующий код:

```
.code
     mov ecx, 0             ; инициализация счетчика цикла
L1:
     cmp ecx, 10            ; сравнение счетчика с размером массива
     jge exit               ; выход из цикла, если счетчик >= 10
     mov eax, array[ecx]    ; загрузка элемента массива по индексу ECX в
                            ; регистр EAX
     add eax, 1             ; увеличение элемента на 1
     mov array[ecx], eax    ; сохранение измененного элемента обратно в
                            ; массив
     inc ecx                ; увеличение счетчика цикла
     jmp L1                 ; переход на следующую итерацию цикла
exit:
```

Для выполнения различных операций над элементами массива могут использоваться различные инструкции, такие как **ADD**, **SUB**, **MUL**, **DIV** и др.

Также существуют специальные инструкции для работы с массивами, такие как **LODS** и **STOS**, которые позволяют загружать и сохранять данные из (в) памяти автоматически, используя указатель на массив. Например, для загрузки каждого элемента массива в регистр *EAX* и увеличения его на 1 можно использовать следующий код:

```
.code
     mov esi, OFFSET array  ; загрузка указателя на массив в регистр ESI
     mov ecx, 10            ; инициализация счетчика цикла
L1:
     cmp ecx, 0             ; сравнение счетчика с 0
     jle exit               ; выход из цикла, если счетчик <= 0
```

```
    lodsd                   ; загрузка элемента массива в регистр EAX и
                            ; увеличение указателя на 4 байта
    add eax, 1              ; увеличение элемента на 1
    stosd                   ; сохранение измененного элемента обратно в
                            ; массив и увеличение указателя на 4 байта
    dec ecx                 ; уменьшение счетчика цикла
    jmp L1                  ; переход на следующую итерацию цикла
exit:
```

В данном примере используются инструкции **LODSD** и **STOSD**. Инструкция **LODSD** загружает 32-битное значение из памяти по адресу, который хранится в регистре *ESI*, и увеличивает значение регистра на 4 байта. Инструкция **STOSD** сохраняет 32-битное значение из регистра *EAX* в памяти по адресу, который хранится в регистре *EDI*, и увеличивает значение регистра на 4 байта.

Для работы с многомерными массивами можно использовать инструкцию **LEA** (*Load Effective Address*), которая загружает в регистр адрес элемента массива по его индексам. Например, для загрузки адреса элемента массива с индексами [3], [4] в регистр *EAX* можно использовать следующий код:

```
lea eax, array[3*4*4+4*4]   ; загрузка адреса элемента массива
по                          ; индексам [3][4]
```

В данном примере используется размер элемента массива 4 байта (тип *DWORD*), поэтому для получения адреса элемента с индексами [3], [4] необходимо умножить индексы на соответствующие коэффициенты и сложить результаты.

Также для работы с массивами можно использовать различные инструкции условного и безусловного перехода, такие как **JMP** (безусловный переход), **JE** (переход, если равно), **JNE** (переход, если не равно), **JG** (переход, если больше) и т. д.

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

Для обращения к элементу массива используют **номер строки** `row_index` и **номер столбца** `column_index`:

```
.data
table       DWORD       10h, 20h, 30h, 40h, 50h
```

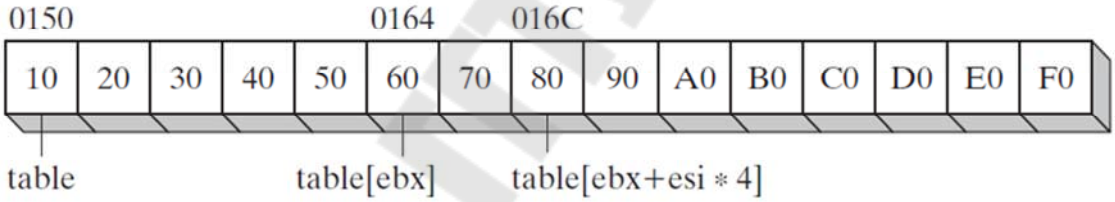
```
RowSize = ($ - table)
          DWORD       60h, 70h, 80h, 90h, 0A0h
          DWORD       0B0h, 0C0h, 0D0h, 0E0h, 0F0h

.code
      row_index = 1                       ; номер строки
      column_index  = 2                   ; номер столбца
      mov ebx, OFFSET table               ; адрес начала массива
      add ebx, RowSize * row_index        ; смещение по строкам
      mov esi, column_index               ; смещение по столбцам
      mov eax, [ebx + esi * TYPE table] ; в EAX загружено значение 80
```

Символ «$» в Ассемблере MASM32 представляет текущий адрес в памяти.

При использовании ***базово-индексной со смещением адресации*** смещение может быть постоянным выражением или именем переменной (массива), ***базовый операнд*** может содержать смещение строки, а ***индексный операнд*** – смещение столбца:

[*база* + *индекс* + *смещение*]

смещение [*база* + *индекс*]




```
.data
table       DWORD       10h, 20h, 30h, 40h, 50h
RowSize = ($ - table)
          DWORD       60h, 70h, 80h, 90h, 0A0h
          DWORD       0B0h, 0C0h, 0D0h, 0E0h, 0F0h

.code
      mov ebx, RowSize                        ; длина строки
      mov esi, 2                              ; номер столбца
      mov eax, table[ebx + esi * TYPE table]  ; EAX <- 80h
```

MOVSB, **MOVSD** и **MOVSW** – это команды Ассемблера для копирования данных из одной области памяти в другую. В MASM32 они используются для копирования байтов, 32-битных слов и 16-битных слов соответственно.

Команда **MOVSB** копирует один байт из одного места в другое. Данная команда копирует байт из адреса, который находится в регистре *SI*, в адрес, который находится в регистре *DI*, и затем инкрементирует значения регистров *SI* и *DI* на 1. Это может быть использовано для копирования одиночных байтов, например, для копирования символов из одной строки в другую.

Команда **MOVSD** копирует 32-битное слово из одного места в другое. Данная команда копирует 32-битное слово из адреса, который находится в регистре *ESI*, в адрес, который находится в регистре *EDI*, и затем инкрементирует значения регистров *ESI* и *EDI* на 4. Это может быть использовано для копирования массивов или структур данных.

Команда **MOVSW** копирует 16-битное слово из одного места в другое. Данная команда копирует 16-битное слово из адреса, который находится в регистре *SI*, в адрес, который находится в регистре *DI*, и затем инкрементирует значения регистров *SI* и *DI* на 2. Это может быть использовано для копирования массивов или структур данных, которые содержат 16-битные элементы.

REP, **REPZ** (или **REPE**) и **REPNZ** (или **REPNE**) – это префиксы команд процессора, которые используются в MASM32 для повторения некоторых инструкций.

Префикс **REP** используется для повторения инструкции, которая следует за ним, пока значение регистра *ECX* не станет равным нулю. Например, команда REP MOVSB скопирует блок памяти из одного места в другое, пока регистр *ECX* не станет равным нулю.

Префикс **REPZ** (или **REPE**) используется для повторения инструкции, которая следует за ним, пока значение регистра *ECX* не станет равным нулю или пока значение флага *ZF* (*Zero Flag*) не станет равным 0. Например, команда `REPZ SCASB` будет повторять инструкцию, пока регистр *ECX* не станет равным нулю или пока не будет найден символ, соответствующий значению AL.

Префикс **REPNZ** (или **REPNE**) используется для повторения инструкции, которая следует за ним, пока значение регистра *ECX* не станет равным нулю или пока значение флага *ZF* не станет равным 1. Например, команда `REPNZ SCASB` будет повторять инструкцию, пока регистр *ECX* не станет равным нулю или пока будет найден символ, не соответствующий значению AL.

Префиксы **REP**, **REPZ** и **REPNZ** наиболее часто используются в сочетании с операцией поиска или сравнения в строках, таких как `SCASB`, `SCASD` или `CMPSB`, `CMPSD`. Они также могут использоваться для повторения других инструкций, которые могут выполняться многократно.

Пример 1. Копировать массив SRS из 50 двойных слов в массив DST

```
.data
op2   DWORD    50 dup (0ffh)       ; инициализация массива SRS[50]
op1   DWORD    50 dup (?)          ; объявление массива DST[50]

.code
      cld                          ; задаем увеличение ESI, EDI на 4
      mov esi, OFFSET op2          ; ESI <- адрес SRS
      mov edi, OFFSET op1          ; EDI <- адрес DST
      mov ecx, LENGTHOF op2        ; ECX <- число элементов SRS
      rep movsd                    ; копирование SRS[i] -> DST[i]
```

CMPSB, **CMPSW** и **CMPSD** – это команды Ассемблера, используемые в MASM32 для сравнения данных в двух областях памяти.

Команда **CMPSB** сравнивает два байта, один из которых находится в памяти по адресу, указанному в регистре *SI*, а второй – по адресу, указанному в регистре *DI*. Данная команда сравнивает байты, уменьшает значения регистров *DI* и *SI* на 1 и устанавливает флаги, основанные на результате сравнения. Это может быть использовано для проверки двух строк на равенство или для поиска определенного символа в строке.

Команда **CMPSW** сравнивает два 16-битных слова, одно из которых находится в памяти по адресу, указанному в регистре *SI*, а второе – по адресу, указанному в регистре *DI*. Данная команда сравнивает 16-битные слова, уменьшает значения регистров *DI* и *SI* на 2 и устанавливает флаги, основанные на результате сравнения. Это может быть использовано для проверки двух строк на равенство или для поиска определенного слова в строке.

Команда **CMPSD** сравнивает два 32-битных слова, одно из которых находится в памяти по адресу, указанному в регистре *SI*, а второе – по адресу, указанному в регистре *DI*. Данная команда сравнивает 32-битные слова, уменьшает значения регистров *DI* и *SI* на 4 и устанавливает флаги, основанные на результате сравнения. Это может быть использовано для проверки двух строк на равенство или для поиска определенного 32-битного значения в строке.

Все три команды используют флаги процессора для определения результата сравнения. Флаги, такие как *CF*, *ZF*, *SF* и *OF*, могут быть использованы для принятия решения об основном результате сравнения.

Пример 2. Сравнить побайтово две строки.

```
.data
array1   byte   “memorax”
array2   byte   “memorex”

.code
     cld                                ; увеличение ESI и EDI на 1
     mov esi, OFFSET array1             ; ESI <- адрес array1
     mov edi, OFFSET array2             ; EDI <- адрес array2
     mov ecx, LENGTHOF array1           ; ECX <- длина array1
     repe cmpsb                         ; побайтное сравнение пока
                                        ; array1[i] = array2[i]
     cmp ecx, 0                         ; если ECX = 0, то строки равны
     jnz m1                             ; иначе переходим на m1
     <вывод: «стрОки совпадают»>
     jmp m2
m1:
     <вывод: «стрОки не совпадают»>
m2:
```

Рассмотрим пример программы нахождения суммы элементов двумерного массива, которые расположены на нечетных строках и столбцах.

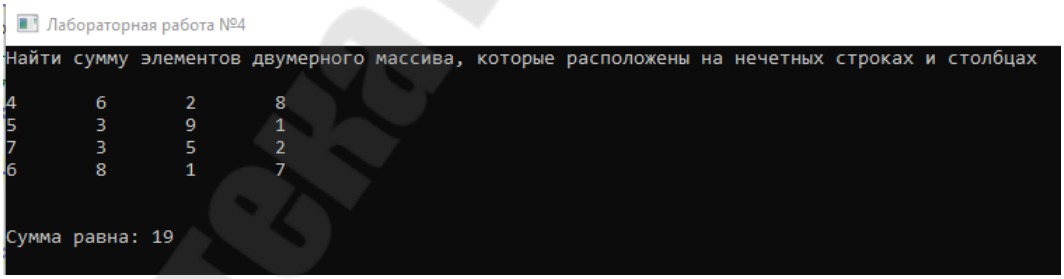
```
.686
.model flat, stdcall                    ; определяем модель памяти и
                                        ; модель вызова функций
option casemap:none                     ; отключаем регистрозависимость

; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\windows.inc
include C:\masm32\include\user32.inc
include C:\masm32\include\kernel32.inc
include C:\masm32\include\masm32.inc

includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\masm32.lib
; Сегмент данных
;-----------------------------------------
.data
sConsTitle BYTE       "Лабораторная работа №4", 0
```

```
strTask     BYTE        "Найти сумму элементов двумерного массива, кото-
рые
                        расположены на нечетных строках и столбцах", 0Dh,
                        0Ah, 0Ah, 0
Arr         DWORD       4, 6, 2, 8          ; двумерный массив 4x4
RowSize = ($ - Arr)
            DWORD 5, 3, 9, 1
            DWORD 7, 3, 5, 2
            DWORD 6, 8, 1, 7

typeArr     BYTE        TYPE Arr            ; размер элемента в байтах
row         BYTE        4                   ; число строк
col         BYTE        4                   ; число столбцов
i           BYTE        0                   ; индекс строки
j           BYTE        0                   ; индекс столбца
sum         DWORD       0                   ; переменная для хранения суммы
resultText  BYTE        "Сумма равна: "
resultStr   BYTE        20 DUP(?), 0
buffer      BYTE        20 DUP(?), 0
tab         BYTE        '     ', 0
clrt        BYTE        0Ah, 0Dh, 0

; Сегмент кода
;-----------------------------------------
.code
start:
; вывод заголовка консоли
      invoke SetConsoleTitle, ADDR sConsTitle
      invoke StdOut, ADDR strTask

; вывод массива
      xor ecx, ecx                  ; ECX = 0
      mov cl, row                   ; ECX = row - загружаем счетчик строк
      mov esi, 0                    ; ESI = 0  – указатель на нулевую
строку
loop_row_1:
      push ecx                      ; сохраняем счетчик внешнего цикла в
стек  xor ecx, ecx                  ; ECX = 0
      mov cl, col                   ; ECX = col - загружаем счетчик столб-
цов
      mov ebx, 0                    ; EBX = 0 - указатель на нулевой стол-
бец
loop_col_1:
      mov eax, Arr[esi][ebx]        ; EAX <- Arr[i][j]
      push ebx                      ; сохраняем регистры перед
```

```
                                ; использованием ltoa и StdOut
      push ecx
      push esi
      invoke ltoa, eax, ADDR buffer
                                ; преобразование Arr[i][j] в строку
      invoke StdOut, ADDR buffer  ; вывод Arr[i][j]
      invoke StdOut, ADDR tab     ; вывод TAB
      pop esi                   ; восстанавливаем регистры из стека
      pop ecx
      pop ebx
      add ebx, TYPE Arr         ; увеличиваем указатель столбцов на
                                ; размер одного элемента
      loop loop_col_1           ; проверяем окончание строки
      invoke StdOut, ADDR clrt    ; переходим на следующую строку

      add esi, RowSize          ; увеличиваем указатель строки на
                                ; размер одной строки
      pop ecx                   ; восстанавливаем счетчик внешнего
цикла
      loop loop_row_1           ; проверяем окончание массива
;-----------------------------------------
; вычисление суммы
      xor ecx, ecx              ; ECX = 0
      mov cl, row               ; ECX = row - загружаем счетчик строк
      mov esi, 0                ; ESI = 0   - указатель на нулевую
строку
      mov i, 0
loop_row_2:
      push ecx                ; сохраняем счетчик внешнего цикла в
стек
      xor ecx, ecx              ; ECX = 0
      mov cl, col               ; ECX = col - загружаем счетчик столб-
цов
      mov ebx, 0                ; EBX = 0 - указатель на нулевой стол-
бец
      mov j, 0
loop_col_2:
      mov eax, Arr[esi][ebx]    ; EAX <- Arr[i][j]
      test i, 1                 ; проверяем является ли строка нечет-
ной
      jz skip_row               ; если нет, то переходим по метке
      test j, 1                 ; проверяем является ли столбец
                                ; нечетным
      jz skip_col               ; если нет, то переходим по метке
      add sum, eax              ; добавляем текущий элемент к сумме
```

```
skip_col:
      inc j                         ; увеличиваем индекс столбца
      add ebx, TYPE Arr             ; увеличиваем указатель столбцов на
                                    ; размер одного элемента
      loop loop_col_2               ; проверяем окончание строки
      invoke StdOut, ADDR clrt      ; переходим на следующую строку
skip_row:
      inc i                         ; увеличиваем индекс строки
      add esi, RowSize              ; увеличиваем указатель строки на
                                    ; размер одной строки
      pop ecx                       ; восстанавливаем счетчик внешнего
цикла
      loop loop_row_2               ; проверяем окончание массива
;-----------------------------------------
; вывод результата
      invoke ltoa, sum, ADDR resultStr        ; преобразование SUM в
строку
      invoke StdOut, ADDR resultText          ; вывод результата

      invoke Sleep, INFINITE
; завершение процессов Windows

      invoke ExitProcess, NULL
; окончание сегмента start
end start
```

Результат тестирования программы:



Лабораторная работа № 6

Цель работы: освоить работу с многомерными массивами в Ассемблере, научиться применять их при решении задач.

Задания для лабораторной работы

1. Составить блок-схему алгоритма и написать текст программы в соответствии с индивидуальным заданием. Индивидуальные задания, в соответствии с номером студента в журнале группы, приведены в табл. 6.1.

Таблица 6.1

Варианты заданий к лабораторной работе № 6

Номер	Задание
1	Найти максимальный элемент в двумерном массиве и вернуть его координаты
2	Проверить, является ли двумерный массив симметричным относительно своей побочной диагонали
3	Проверить, является ли двумерный массив симметричным относительно своей главной диагонали
4	Найти сумму элементов двумерного массива, которые находятся ниже побочной диагонали
5	Найти сумму элементов двумерного массива, которые находятся ниже главной диагонали
6	Поменять местами столбцы двумерного массива по заданным индексам
7	Найти сумму элементов двумерного массива, которые находятся на побочной диагонали
8	Проверить, является ли первый столбец двумерного массива упорядоченным по убыванию
9	Проверить, является ли вторая строка двумерного массива упорядоченной по убыванию
10	Найти сумму максимальных элементов каждой строки двумерного массива
11	Найти минимальный элемент в каждой строке двумерного массива
12	Посчитать сумму элементов в каждом столбце двумерного массива
13	Проверить, является ли каждый элемент двумерного массива четным
14	Найти количество четных элементов в двумерном массиве
15	Поменять знак у всех элементов двумерного массива
16	Проверить, является ли каждый элемент двумерного массива положительным
17	Найти наименьший положительный элемент в двумерном массив
18	Проверить, является ли каждый элемент двумерного массива простым числом
19	Найти количество простых чисел в двумерном массиве
20	Найти сумму элементов двумерного массива, которые делятся на 3
21	Найти все индексы заданного элемента в двумерном массиве
22	Найти индексы первого вхождения заданного элемента в двумерном массиве

Окончание табл. 6.1

Номер	Задание
23	Найти произведение элементов двумерного массива, которые делятся на 2
24	Найти наибольший отрицательный элемент в двумерном массиве
25	Найти максимальный из элементов двумерного массива, которые расположены на четных строках и столбцах
26	Проверить, является ли каждый элемент двумерного массива уникальным
27	Найти наименьший неуникальный элемент в двумерном массиве
28	Проверить, является ли двумерный массив квадратным (все строки и столбцы имеют одинаковую длину)
29	Найти максимальный элемент в каждой строке двумерного массива
30	Найти максимальный из элементов двумерного массива, расположенных на главной диагонали

2. Открыть исполняемый файл программы в отладчике. Выполнить программу в пошаговом режиме. После выполнения каждого шага заносить данные в табл. 6.2.

Таблица 6.2

Результат выполнения программы в отладчике*

Регистры				Флаги				Память
EAX	*EBX*	…	…	*FC*	*FZ*	…	…	

*В таблицу должны быть включены те элементы (регистры, флаги и пр.), которые имеют место в конкретном варианте программы.

3. Выполнить тесты программы

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Блок-схема алгоритма.
5. Текст программ с комментариями.
6. Результаты выполнения программы в пошаговом режиме в отладчике.
7. Скриншоты тестов программы.
8. Выводы.

Лабораторная работа № 6а

Цель работы: освоить работу с массивами данных в Ассемблере, научиться применять их при решении задач сортировки массивов.

Задания для лабораторной работы

1. Составить блок-схему алгоритма и написать текст программы в соответствии с индивидуальным заданием. Индивидуальные задания, в соответствии с номером студента в журнале группы, приведены в табл. 6.3.

Таблица 6.3

Задания для лабораторной работы № 6а

Вариант	Задание
01	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $D_1 \geq D_2 \geq ... \geq D_M$, где D_i – максимальное значение среди всех элементов *i*-й строки
02	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $C_1 \leq C_2 \leq ... \leq C_N$, где C_j – минимальное значение среди всех элементов *j*-го столбца
03	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $F_1 \geq F_2 \geq ... \geq F_M$, где F_i – сумма всех элементов *i*-й строки
04	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $P_1 \leq P_2 \leq ... \leq P_N$, где P_j – произведение всех элементов *j*-го столбца
05	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $K_1 \geq K_2 \geq ... \geq K_N$, где K_j – количество положительных элементов в *j*-м столбце
06	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $L_1 \leq L_2 \leq ... \leq L_M$, где L_i – количество отрицательных элементов в *i*-й строке
07	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $R_1 \geq R_2 \geq \ldots \geq R_N$, где R_j – количество нулевых элементов в *j*-м столбце
08	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $S_1 \leq S_2 \leq ... \leq S_M$, где S_i – сумма абсолютных значений всех элементов *i*-й строк
09	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $T_1 \leq T_2 \leq ... \leq T_N$, где T_j – максимальное значение среди всех элементов *j*-го столбца

Продолжение табл. 6.3

Вариант	Задание
10	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $Q_1 \geq Q_2 \geq ... \geq Q_M$, где Q_i – максимальное значение среди всех элементов *i*-й строки
11	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $Z_1 \leq Z_2 \leq ... \leq Z_N$, где Z_j – сумма всех элементов *j*-го столбца
12	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $X_1 \geq X_2 \geq ... \geq X_M$, где X_i – произведение всех элементов *i*-й строки
13	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $H_1 \leq H_2 \leq ... \leq H_M$, где H_i – количество положительных элементов в *i*-й строке
14	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $A_1 \geq A_2 \geq ... \geq A_N$, где A_j – количество отрицательных элементов в *j*-м столбце
15	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $B_1 \leq B_2 \leq ... \leq B_M$, где B_i – количество нулевых элементов в *i*-й строке
16	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $C_1 \geq C_2 \geq ... \geq C_N$, где C_j – сумма абсолютных значений всех элементов *j*-го столбца
17	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $Y_1 \leq Y_2 \leq ... \leq Y_M$, где Y_i – максимальное значение среди всех элементов *i*-й строки
18	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $E_1 \geq E_2 \geq ... \geq E_N$, где E_j – минимальное значение среди всех элементов *j*-го столбца
19	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $F_1 \leq F_2 \leq ... \leq F_M$, где F_i – сумма всех элементов *i*-й строки
20	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $G_1 \geq G_2 \geq ... \geq G_N$, где G_j – произведение всех элементов *j*-го столбца
21	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $L_1 \geq L_2 \geq ... \geq L_M$, где L_i – количество положительных элементов в *i*-й строке
22	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $T_1 \leq T_2 \leq ... \leq T_N$, где T_j – количество отрицательных элементов в *j*-м столбце

Окончание табл. 6.3

Вариант	Задание
23	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $K_1 \geq K_2 \geq ... \geq K_M$, где K_i – количество нулевых элементов в *i*-й строке
24	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $P_1 \leq P_2 \leq ... \leq P_N$, где P_j – сумма абсолютных значений всех элементов *j*-го столбца
25	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $X_1 \geq X_2 \geq ... \geq X_N$, где X_j – максимальное значение среди всех элементов *j*-го столбца
26	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $Y_1 \leq Y_2 \leq ... \leq Y_M$, где Y_i – минимальное значение среди всех элементов *i*-й строки
27	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $E_1 \geq E_2 \geq ... \geq E_N$, где E_j – сумма всех элементов *j*-го столбца
28	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $V_1 \leq V_2 \leq ... \leq V_M$, где V_i – произведение всех элементов *i*-й строки
29	В матрице *M*x*N* переставить столбцы таким образом, чтобы получилась последовательность $W_1 \leq W_2 \leq ... \leq W_N$, где W_j – количество положительных элементов в *j*-м столбце
30	В матрице *M*x*N* переставить строки таким образом, чтобы получилась последовательность $Z_1 \geq Z_2 \geq ... \geq Z_M$, где Z_i – количество отрицательных элементов в *i*-й строке

2. Выполнить тесты программы.

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Блок-схема алгоритма.
5. Текст программ с комментариями.
6. Скриншоты тестов программы.
7. Выводы.

Контрольные вопросы

1. Что такое массив в программировании?
2. Как объявить массив в MASM32?

3. Как получить доступ к элементу массива по его индексу?
4. Как узнать размер (количество элементов) массива?
5. Как заполнить массив значениями?
6. Как выполнить обход (итерацию) всех элементов массива?
7. Как найти наибольший элемент в массиве?
8. Как отсортировать массив в порядке возрастания или убывания?
9. Как скопировать содержимое одного массива в другой?

Глава 7. Структуры данных

Структуры в MASM32 – это набор связанных переменных разных типов, объединенных под общим именем. Они позволяют удобно хранить и обрабатывать связанные между собой данные.

Для создания структуры в MASM32 используется директива **STRUCT**, после которой идет список полей, разделенных запятой. Каждое поле задается в формате «**имя поля: тип поля**». Типы полей могут быть любыми базовыми типами, такими как BYTE, WORD, DWORD, а также другими структурами.

Пример создания структуры «Книга» с четырьмя полями:

```
Book STRUCT
    Title   DWORD   ?
    Author  DWORD   ?
    Year    DWORD   ?
    Pages   DWORD   ?
Book ENDS
```

После определения структуры можно создавать переменные, используя ее имя в качестве типа. Для доступа к полям структуры используется оператор точка (.), с помощью которого можно получить доступ к любому полю.

Пример создания переменной типа «Книга» и доступа к ее полям:

```
; создание переменной MyBook типа Book
    MyBook Book <>, <>, <>, <>
; инициализация полей структуры MyBook
    mov MyBook.Title, OFFSET BookTitle
    mov MyBook.Author, OFFSET BookAuthor
    mov MyBook.Year, OFFSET BookYear
    mov MyBook.Pages, OFFSET BookPages
```

```
; доступ к полям структуры MyBook
    mov eax, MyBook.Title
    mov ebx, MyBook.Author
    mov ecx, MyBook.Year
    mov edx, MyBook.Pages
```

Структуры в MASM32 могут быть использованы для создания сложных типов данных, таких как списки, очереди, стеки и др. Они также могут быть использованы для передачи параметров в функции и возвращения значений из функций.

Пример использования структуры для передачи параметров в функцию:

```
; определение функции, принимающей параметр типа "Book"

MyFunction PROC BookParam:PTR Book
    ; доступ к полям структуры через переданный указатель
    mov eax, [BookParam].Title
    mov ebx, [BookParam].Author
    mov ecx, [BookParam].Year
    mov edx, [BookParam].Pages
    ret
MyFunction ENDP

; вызов функции и передача параметра типа "Книга"
invoke MyFunction, OFFSET MyBook
```

Таким образом, структуры предоставляют удобный способ для организации и обработки данных в MASM32.

Пример вывода элементов структуры:

```
.data
MyStruct STRUCT             ; определяем структуру MyStruct
     x  DWORD  ?                  ; поле x типа DWORD
     y  DWORD  ?                  ; поле y типа DWORD
     z  DWORD  ?                  ; поле z типа DWORD
MyStruct ENDS
myVar MyStruct <1, 2, 3>          ; определяем переменную myVar типа
                                       ; MyStruct и инициализируем ее
MyStruct_num  EQU  SIZEOF MyStruct / TYPE DWORD

.code
```

```
start:
      mov ecx, MyStruct_num
      mov esi, OFFSET myVar
for_loop:
      push ecx                          ; сохраняем счетчик в стек
      mov eax, [esi].MyStruct.x         ; получение значения поля струк-
туры
      invoke ltoa, eax, ADDR buffer     ; преобразование в строку
      invoke StdOut, ADDR buffer        ; вывод
      add esi, TYPE DWORD          ; переходим к следующему полю структу-
ры
      pop ecx                      ; извлекаем счетчик из стека
      loop for_loop                ; завершаем итерацию
```

Пример нахождение суммы полей структуры:

```
.data
MyStruct  STRUCT                   ; определяем структуру MyStruct
      x  DWORD  ?                  ; поле x типа DWORD
      y  DWORD  ?                  ; поле y типа DWORD
      z  DWORD  ?                  ; поле z типа DWORD
MyStruct  ENDS
myVar  MyStruct  <1, 2, 3>         ; определяем переменную myVar типа
                                        ; MyStruct и инициализируем ее

.code
start:
      mov eax, myVar.x             ; myVar.x  -> EAX
      add eax, myVar.y             ; EAX = myVar.x + myVar.y
      eax, myVar.z                 ; EAX = myVar.x + myVar.y + myVar.z

      invoke ExitProcess, 0        ; завершаем программу
end start
```

Когда мы говорим о структурах в MASM32, мы имеем в виду набор связанных данных, которые могут быть разных типов, хранящихся вместе в определенном порядке.

Чтобы передать структуру в процедуру, мы не можем просто передать значения ее полей, потому что это займет много места и будет неэффективно. Вместо этого мы можем передать адрес (указатель) на структуру, что позволяет нам получить доступ к ее полям и изменять их.

Когда мы передаем указатель на структуру в процедуру, мы передаем адрес первого поля структуры. Это позволяет процедуре получить доступ к остальным полям структуры, используя смещения относительно адреса первого поля.

Для передачи указателя на структуру в качестве параметра процедуры в MASM32 мы используем команду **push**, чтобы поместить адрес в стек, и затем вызываем процедуру. Внутри процедуры мы объявляем параметр как указатель на структуру, который будет содержать адрес переданной структуры.

Затем мы можем использовать этот указатель для доступа к полям структуры и изменения их значений внутри процедуры.

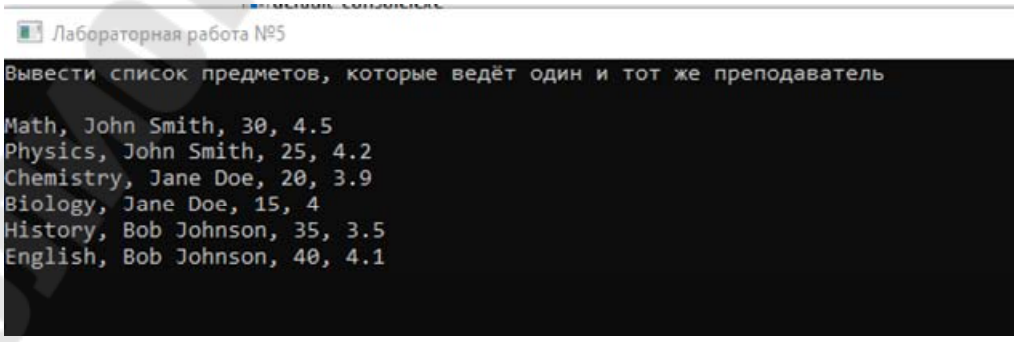
В приведенном ниже примере объявлена структура **Subject** и создано шесть переменных типа **Subject**. Далее для вывода полей структуры в консоль ипользуется процедура (подпрограмма) **PrintStruct**. При каждом вызове процедуры **PrintStruct** в нее передается указатель на ту переменную, поля которой выводятся.

```
.686
.model flat, stdcall
option casemap:none

; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\windows.inc
include C:\masm32\include\user32.inc
include C:\masm32\include\kernel32.inc
include C:\masm32\include\masm32.inc
include C:\masm32\include\msvcrt.inc

includelib C:\masm32\lib\user32.lib
includelib C:\masm32\lib\kernel32.lib
includelib C:\masm32\lib\masm32.lib
includelib C:\masm32\lib\msvcrt.lib

; Объявление структуры
Subject STRUCT
discipline BYTE  20 DUP(?)             ; название предмета
teacher    BYTE  20 DUP(?)             ; преподаватель
    student_num  DWORD      ?          ; число студентов
    avg_grade    QWORD      ?          ; средний балл
Subject ENDS
```

```
; Прототип процедуры вывода структуры
PrintStruct PROTO ptrSubject:PTR Subject

; Сегмент данных
;-----------------------------------------
.data
sConsTitle BYTE  "Лабораторная работа №5", 0
strTask    BYTE  "Вывести список предметов, которые ведет один и тот же
"
           BYTE  "преподаватель.", 0Dh, 0Ah, 0Ah, 0
clrt       BYTE  0Ah, 0Dh, 0                  ; перевод каретки
comma BYTE  ", ", 0
buffer     BYTE  20 DUP(0)                    ; буфер для преобразования

; создание переменных типа Subject
subject_1  Subject    <"Math", "John Smith", 30, 4.5>
subject_2  Subject    <"Physics", "John Smith", 25, 4.2>
subject_3  Subject    <"Chemistry", "Jane Doe", 20, 3.9>
subject_4  Subject    <"Biology", "Jane Doe", 15, 4.0>
subject_5  Subject    <"History", "Bob Johnson", 35, 3.5>
subject_6  Subject    <"English", "Bob Johnson", 40, 4.1>

; Сегмент кода
;-----------------------------------------
.code

main PROC
; вывод заголовка консоли
      invoke SetConsoleTitle, ADDR sConsTitle
      invoke StdOut, ADDR strTask

; вывод полей структур в консоль
      invoke PrintStruct, ADDR subject_1
      invoke PrintStruct, ADDR subject_2
      invoke PrintStruct, ADDR subject_3
      invoke PrintStruct, ADDR subject_4
      invoke PrintStruct, ADDR subject_5
      invoke PrintStruct, ADDR subject_6

      invoke Sleep, INFINITE
; завершение процессов Windows
      invoke ExitProcess, NULL
main ENDP
```

```
;-----------------------------------------
; Процедура вывода структуры
;-----------------------------------------
PrintStruct PROC ptrSubject:PTR Subject   ; передаем указатель на струк-
туру
; вывод поля "discipline"
      mov ebx, ptrSubject
      invoke StdOut, ebx
      invoke StdOut, ADDR comma

; вывод поля "teacher"
      add ebx, SIZEOF Subject.discipline
      invoke StdOut, ebx
      invoke StdOut, ADDR comma

; вывод поля "student_num"
      add ebx, SIZEOF Subject.teacher
      invoke ltoa, [ebx], ADDR buffer
      invoke StdOut, ADDR buffer
      invoke StdOut, ADDR comma

; вывод поля "avg_grade"
      add ebx, SIZEOF Subject.student_num
      invoke FloatToStr, [ebx], ADDR buffer
      invoke StdOut, ADDR buffer
      invoke StdOut, ADDR clrt

      ret
PrintStruct ENDP
;-----------------------------------------

END main
```

Результат вывода переменных представлен ниже:



Лабораторная работа № 7

Цель работы: освоить работу со структурами данных в Ассемблере, научиться применять их при решении задач.

Задания для лабораторной работы

1. Составить блок-схему алгоритма и написать текст программы в соответствии с индивидуальным заданием. Индивидуальные задания, в соответствии с номером студента в журнале группы, приведены в табл. 7.1.

Таблица 7.1

Задания к лабораторной работе № 7

Вариант	Задание
01	Создайте структуру «Студент» с полями «имя», «фамилия», «возраст» и «средний балл». Напишите процедуру, которая выводит список студентов, у которых средний балл выше заданного порога
02	Создайте структуру «Заказ» с полями «номер», «дата», «количество» и «цена». Напишите процедуру, которая выводит общую стоимость заказов за заданный период
03	Создайте структуру «Работник» с полями «имя», «фамилия», «должность» и «зарплата». Напишите процедуру, которая выводит список работников, у которых зарплата выше заданной суммы
04	Создайте структуру «Книга» с полями «название», «автор», «издательство» и «год издания». Напишите процедуру, которая выводит список книг, изданных за заданный период
05	Создайте структуру «Клиент» с полями «имя», «фамилия», «адрес» и «номер телефона». Напишите процедуру, которая выводит список клиентов, чьи фамилии начинаются с заданной буквы
06	Создайте структуру «Товар» с полями «название», «описание», «цена» и «количество на складе». Напишите процедуру, которая выводит список товаров, цена которых ниже заданной
07	Создайте структуру «Адрес» с полями «улица», «дом», «квартира» и «индекс». Напишите процедуру, которая выводит список адресов, находящихся в заданном городе
08	Создайте структуру «Фильм» с полями «название», «жанр», «режиссер» и «год выпуска». Напишите процедуру, которая выводит список фильмов, выпущенных в заданном году
09	Создайте структуру «Счет» с полями «номер», «дата», «сумма» и «клиент». Напишите процедуру, которая выводит список счетов, открытых у заданного клиента

Продолжение табл. 7.1

Вариант	Задание
10	Создайте структуру «Автор» с полями «имя», «фамилия», «годы жизни» и «страна». Напишите процедуру, которая выводит список авторов, чьи фамилии начинаются с заданной буквы
11	Создайте структуру «Компьютер» с полями «марка», «модель», «цена» и «операционная система». Напишите процедуру, которая выводит список компьютеров, цена которых находится в заданном диапазоне
12	Создайте структуру «Здание» с полями «адрес», «владелец», «площадь» и «количество этажей». Напишите процедуру, которая выводит список зданий, принадлежащих заданному владельцу
13	Создайте структуру «Пациент» с полями «имя», «фамилия», «дата рождения» и «диагноз». Напишите процедуру, которая выводит список пациентов, с диагнозом, совпадающим с заданным
14	Создайте структуру «Продукт» с полями «название», «описание», «цена» и «категория». Напишите процедуру, которая выводит список продуктов, принадлежащих заданной категории
15	Создайте структуру «Курс» с полями «название», «преподаватель», «дни недели» и «время занятий». Напишите процедуру, которая выводит список курсов, проводимых в заданный день недели
16	Создайте структуру «Автомобиль» с полями «марка», «модель», «год выпуска» и «цена». Напишите процедуру, которая выводит список автомобилей, цена которых выше заданной суммы
17	Создайте структуру «Файл» с полями «имя», «расширение», «размер» и «дата создания». Напишите процедуру, которая выводит список файлов, созданных в заданный период
18	Создайте структуру «Поставщик» с полями «имя», «адрес», «номер телефона» и «электронная почта». Напишите процедуру, которая выводит список поставщиков, чьи имена начинаются с заданной буквы
19	Создайте структуру «Предмет» с полями «название», «преподаватель», «количество студентов» и «средний балл». Напишите процедуру, которая выводит список предметов, на которых средний балл выше заданного порога
20	Создайте структуру «Страна» с полями «название», «столица», «население» и «язык». Напишите процедуру, которая выводит список стран, чьи названия начинаются с заданной буквы
21	Создайте структуру «Банк» с полями «название», «адрес», «номер телефона» и «электронная почта». Напишите процедуру, которая выводит список банков, чьи названия начинаются с заданной буквы

Окончание табл. 7.1

Вариант	Задание
22	Создайте структуру «Статья» с полями «название», «автор», «дата публикации» и «количество просмотров». Напишите процедуру, которая выводит список статей, опубликованных в заданный период
23	Создайте структуру «Ресторан» с полями «название», «адрес», «список блюд» и «средний чек». Напишите процедуру, которая выводит список ресторанов с наибольшим средним чеком
24	Создайте структуру «Квартира» с полями «адрес», «площадь», «количество комнат», «цена за квадратный метр» и «общая цена». Напишите процедуру, которая выводит список квартир, отсортированных по общей цене
25	Создайте структуру «Банк» с полями «название», «адрес», «список клиентов» и «баланс». Напишите процедуру, которая выводит список банков с наибольшим балансом
26	Создайте структуру «Аэропорт» с полями «название», «адрес», «список рейсов» и «количество пассажиров». Напишите процедуру, которая выводит список аэропортов с наибольшим количеством пассажиров
27	Создайте структуру «Компания» с полями «название», «адрес», «список сотрудников» и «доход». Напишите процедуру, которая выводит список компаний с наибольшим доходом
28	Создайте структуру «Запись в блоге» с полями «заголовок», «текст», «автор», «дата публикации» и «количество просмотров». Напишите процедуру, которая выводит список всех записей в порядке убывания количества просмотров
29	Создайте структуру «Организация» с полями «название», «адрес», «список сотрудников» и «доход» типа DWORD. Напишите процедуру, которая выводит список всех сотрудников и общий доход организации
30	Создайте структуру «Предмет» с полями «название», «преподаватель», «количество студентов» и «средний балл». Напишите процедуру, которая выводит список предметов, которые ведет один и тот же преподаватель

2. Для каждого задания необходимо создать не менее семи переменных типа заданной структуры, и выполнить над ними необходимое действие.

3. Разработать и выполнить все необходимые тесты программы.

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Блок-схема алгоритма.
5. Текст программ с комментариями.
6. Скриншоты тестов программы.
7. Выводы.

Контрольные вопросы

1. Что такое структура в Ассемблере MASM32?
2. Как объявить структуру в MASM32?
3. Каким образом происходит обращение к полям структуры в MASM32?
4. Каким образом можно проинициализировать структуру в MASM32?
5. Как осуществить доступ к структурам, объявленным в других модулях в MASM32?
6. Как можно определить размер структуры в MASM32?
7. Каким образом можно передать структуру в качестве параметра функции в MASM32?
8. Возможно ли в MASM32 создать структуру, содержащую указатель на функцию? Если да, то как?
9. Каким образом происходит выравнивание структур в MASM32?

Глава 8. Создание библиотечных функций. Работа с файлами

Технология работы с DLL

Чтобы упростить разработку больших программ, создаются библиотеки с часто вызываемыми функциями. Этот процесс называется статической компоновкой. Хорошим примером служат стандартные библиотеки в языке Си. Однако у такого метода есть серьезный недостаток: каждая программа, вызывающая функции из библиотеки, содержит копии этих функций.

Для MS-DOS эта ситуация не была проблемной, поскольку в определенный момент времени могла выполняться только одна программа. Однако в системе Windows в фиксированный промежуток времени могут выполняться десятки и даже сотни различных прило-

жений или процессов (не говоря о многоядерных вычислениях). В этом случае оперативная память будет перегружена.

Все это привело к появлению технологии динамических библиотек DLL. Windows не загружает несколько копий библиотеки в память, а позволяет процессам разделять один и тот же код этой DLL. Впрочем секция данных копируется для каждого процесса.

В отличие от статических библиотек DLL вызывается только в процессе выполнения программы. Библиотеку также можно и выгрузить из памяти во время выполнения программы, если она больше не нужна. Однако память освободится сразу в случае, если все остальные процессы также больше не обращаются к библиотеке.

Перед линкером стоит сложная задача, когда он проводит фиксирование адресов в конечном исполняемом файле. Ему требуется сохранить достаточно информации о DLL и используемых функциях в выходном файле, чтобы тот смог найти и загрузить верную DLL во время выполнения.

Это приводит к необходимости использования библиотеки импорта. Она содержит информацию о DLL, которую представляет. Линкер может получить из нее необходимую информацию и вставить ее в исполняемый файл.

Когда Windows загружает программу в память, она видит, что программа требует DLL, поэтому ищет библиотеку, «мэппирует» ее в адресное пространство процесса и выполняет фиксацию адресов для вызовов функций в DLL.

DLL можно загрузить без использования Windows-загрузчика. Однако с библиотекой импорта это сделать проще.

DLL без точки входа

Рассмотрим способ создания библиотеки на конкретном примере. Пусть требуется создать библиотеку, содержащую функции обработки массивов типа DWORD. В целях упрощения задачи ограничимся лишь одной функцией, вычисляющей сумму элементов массива.

Очевидно, что для обработки процедуре достаточно передать два параметра: адрес массива и количество элементов. Опишем нашу процедуру согласно принятому ранее соглашению:

```
DWORD SumArray([IN] DWORD lpArr,  ; указатель на массив типа
DWORD
               [IN] DWORD Len)    ; число элементов
```

Общий механизм создания и подключения динамической библиотеки следующий:

1. Создается файл с кодом библиотеки.

2. В другом специальном файле указывается, какие функции берутся из библиотеки.

3. Файлы компилируются в библиотеку.

4. Основная программа компилируется со ссылкой на библиотеку.

Создадим код библиотеки в файле **MyDLL.asm**:

```
.686
.model flat, stdcall        ; определяем модель памяти и модель вызова
функций
option casemap :none              ; отключаем регистрозависимость

; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\kernel32.inc
includelib C:\masm32\lib\kernel32.lib

.code
SumArray PROC lpArr: PTR DWORD, len: DWORD
      mov ebx, lpArr          ; EBX <- указатель на массив
                              ; EBX:= указатель на массив
      mov eax, 0              ; в EAX накапливаем сумму
                                    ; в EAX накапливаем сумму
      mov ecx, len            ; ECX <- количество элементов
                                    ; ECX: = количество элементов
@L:
      add eax, [ebx]
      add ebx, 4
      loop @L
; результат в EAX
      ret
SumArray endp
End
```

Обратите внимание, что файл завершен командой **end**, а не **end SumArray**. Это говорит о том, что функция **SumArray** уже ***не выполняет роль точки входа***. Действительно, любая из функций библиотеки может стать точкой входа.

Чтобы другие программы могли вызывать функции библиотеки, требуется описать их в специальном файле **MyDLL.def** экспорта функций:

```
LIBRARY MyDLL
EXPORTS SumArray
```

Директива **LIBRARY** указывает имя библиотеки, а **EXPORTS** – имена экспортируемых функций (т. е. необязательно включать все функции модуля).

Теперь можно скомпилировать оба файла, создав bat-файл:

```
C:\masm32\bin\ml.exe /c /coff MyDLL.asm
C:\masm32\bin\link.exe /DLL /DEF:MyDLL.def /NOENTRY MyDLL.obj
Pause
del MyDLL.exp
del MyDLL.obj
```

– ключ **/DLL** указывает, что компонуется динамическая библиотека;

– ключ **/DEF** ссылается на карту импортируемых функций;

– ключ **/NOENTRY** помечает, что в библиотеке не будет точки входа.

В результате мы получим файл с библиотекой **MyDLL.dll**. Кроме того, автоматически скомпонуется библиотека импорта **MyDLL.lib** и файлы **MyDLL.exp**, **MyDLL.obj** (последние два можно удалить).

Напишем код основной программы:

```
.686
.model flat, stdcall        ; определяем модели памяти и вызова функций
option casemap:none         ; отключаем регистрозависимость

; Библиотеки и подключаемые файлы проекта
;-----------------------------------------
include C:\masm32\include\masm32rt.inc

; подключаем наш файл импорта
includelib D:\Documents\Projects\ASM\LAB_6\MyDLL.lib
; прототип импортируемой функции
SumArray PROTO :PTR DWORD, :DWORD

; Сегмент данных
;-----------------------------------------
.data
      sConsTitle BYTE  "Лабораторная работа №6", 0
```

```
      strTask     BYTE  "Создание библиотек", 0Dh, 0Ah, 0Ah, 0
      Array       DWORD       10, 23, 500, 100, 23, 4
      Sum         DWORD       0
      resultText  BYTE  "Сумма равна: "
      resultStr   BYTE  20 DUP(?), 0

; Сегмент кода
;-----------------------------------------
.code
start:
; вывод заголовка консоли
      invoke SetConsoleTitle, ADDR sConsTitle
      invoke StdOut, ADDR strTask

; вычисление суммы элементов массива
      invoke SumArray, ADDR Array, LENGTHOF Array
      mov Sum, eax

; вывод результата
      invoke ltoa, Sum, ADDR resultStr  ; преобразование SUM в строку
      invoke StdOut, ADDR resultText          ; вывод результата

      invoke Sleep, INFINITE                  ; завершение процессов Win-
dows
      invoke ExitProcess, NULL
; окончание сегмента start
end start
```

При использовании подключаемого файла **masm32rt.inc**, возможно, потребуется дописать полные пути (добавить имя диска **c:**) к файлам библиотек, указанных в нем.

Можно создать заголовочный файл **MyDLL.inc**, в котором описать указанный прототип, и подключить его в программе директивной **include**, чтобы не писать прототипы внутри программы.

DLL с точкой входа

Иногда при работе с библиотекой возникает необходимость проделать операции со всеми процедурами библиотеки (например, задать начальные данные или выделить дополнительную память). В этом случае потребуется создать процедуру, являющуюся точкой входа.

```
.386
.model flat, stdcall
option casemap:none

include windows.inc
include kernel32.inc
includelib kernel32.lib

.code
DLLEntry PROC hInstDLL: HINSTANCE, reason: DWORD, reserved: DWORD
      mov eax, TRUE
      ret
DLLEntry endp

SumArray proc lpArr: PTR DWORD, len: DWORD
;; вычисления ;;
      ret
SumArray endp
end DLLEntry
```

В этой программе **DLLEntry** является точкой входа, причем сама функция реализована специфически. Параметр **hInstDLL** служит дескриптором модуля DLL.

Параметр **reason** может принимать только определенные значения; используется, когда требуется отследить загрузку библиотеки/процесса, чтобы задать начальные значения либо очистить память.

Параметр **reserved** всегда берут нулем (NULL).

Строка

```
mov eax, TRUE
```

важна тем, что позволяет выполняться DLL (если взять его FALSE, то операционная система установит запрет на вызов).

Что касается файла с указанием экспортируемых функций, то функцию DLLEntry указывать здесь не требуется (поскольку она уже является точкой входа).

При компиляции необходимо удалить ключ **/NOENTRY**:

```
c:\masm32\bin\ml /c /coff MyDLL.asm
c:\masm32\bin\link /DLL /SUBSYSTEM:CONSOLE /DEF:MyDLL.def MyDLL.obj
```

Поиск DLL-библиотеки

Операционной системе для успешной загрузки DLL-библиотеки требуется сначала ее найти. Windows осуществляет поиск по следующим ссылкам:

– директория, из которой загружено приложение;

– текущая директория;

– системная директория (обычно C:\Windows\System32);

– системная директория для 16-битных приложений (обычно C:\Windows\System);

– Windows-директория;

– директории, указанные в переменной окружения PATH.

Порядок просмотра директорий может быть иным (зависит от настроек). Если в перечисленных директориях файл не найден, то приложение завершается соответствующим исключением.

Динамическое подключение

Выше был рассмотрен статический способ подключения DLL-библиотек. Это означает, что операционная система автоматически подключает библиотеку к программе. Автоматизация была возможна благодаря использованию библиотек импорта. Поэтому статическое подключение является наиболее простым в плане реализации.

Однако статическое подключение обладает рядом недостатков:

– если библиотека отсутствует, то приложение не сможет быть выполнено и экстренно завершится;

– может отсутствовать файл библиотеки импорта .lib.

Первая проблема очень опасна, поскольку может привести к «краху» приложения. Программисту желательно иметь возможность обработать исключительную ситуацию, по крайней мере осуществив сохранение данных перед экстренным закрытием приложения.

Именно для этих целей используют приемы динамического подключения библиотеки. Обработка осуществляется тремя функциями:

`LoadLibrary` – ищет и загружает библиотеку, в противном случае возвращает 0;

`GetProcAdress` – осуществляет поиск функции в указанной библиотеке и возвращает указатель на нее, иначе 0;

`FreeLibruary` – освобождает библиотеку.

Очевидно, что использование этих функций позволит:

– отследить отсутствие библиотеки и эффективно это обработать;

– отследить и обработать некорректность вызываемой функции;

– освободить ресурсы ОЗУ, если библиотека более не используется.

При динамическом подключении придется «пожертвовать» использованием директивы `INVOKE`: поскольку обращаться к библиотечной функции требуется по указателю, необходимо использовать команду **`call`**.

Код библиотеки и алгоритм компиляции остаются прежними. Код основной программы больше не ссылается на файл с импортом и прототипами.

```
;-----------------------------------------
; Вычислить сумму элементов массива с помощью библиотечной функции
;-----------------------------------------

.686
.model flat, stdcall  ; определяем модели памяти и вызова функций
option casemap:none   ; отключаем регистрозависимость

; Библиотеки и подключаемые файлы проекта
include C:\masm32\include\masm32rt.inc

; Сегмент данных
;-----------------------------------------
.const
   libName     BYTE   "MyDLL", 0h       ; имя библиотеки
   funcName    BYTE   "SumArray", 0h    ; имя функции

;-----------------------------------------
.data
   sConsTitle  BYTE   "Лабораторная работа №6", 0
   strTask     BYTE   "Создание библиотек", 0Dh, 0Ah, 0Ah, 0
   errorMsg    BYTE   "Библиотека не найдена", 0Dh, 0Ah, 0
   Array       DWORD   10, 23, 500, 100, 23, 4
   Sum         DWORD   0
   resultText  BYTE    "Сумма равна: "
   resultStr   BYTE    20 DUP(?), 0

;-----------------------------------------
.data?
   hLib        DWORD   ?           ; дескриптор библиотеки
   lpFunc           DWORD   ?      ; указатель на функцию
```

```
; Сегмент кода
;-----------------------------------------
.code
start:
; вывод заголовка консоли
     invoke SetConsoleTitle, ADDR sConsTitle
     invoke StdOut, ADDR strTask

; -----------------------------------------
; получаем дескриптор. Если вызов успешен (библиотека найдена),
; то сохраняем его в переменную hLib. Иначе - выход с обработ-
кой исключения.
     invoke LoadLibrary, ADDR libName
     cmp eax, 0
     je @error

; -----------------------------------------
; получаем указатель на функцию в библиотеке и вызываем ее, но
уже по адресу!
     mov hLib, eax
     invoke GetProcAddress, hLib, ADDR funcName
     mov lpFunc, eax

     push LENGTHOF Array
     push OFFSET Array
     call [lpFunc]
     mov Sum, eax

;-----------------------------------------
; вывод результата
     invoke ltoa, Sum, ADDR resultStr  ; преобразование SUM в строку
     invoke StdOut, ADDR resultText ; вывод результата

; -----------------------------------------
; Освобождаем ресурсы - больше библиотека не понадобится
     invoke FreeLibrary, hLib
     jmp @exit

@error:
     invoke StdOut, ADDR errorMsg

@exit:
     invoke Sleep, INFINITE
     invoke ExitProcess, NULL ; завершение процессов Windows
end start                     ; окончание сегмента start
```

Динамическое подключение несколько увеличивает объем кода и занимаемой памяти, однако может компенсироваться гибкостью (например, в языке C++ часто используется обращение к функции по указателю на нее).

Функции для работы с файлами

Работа с файлами в MASM32 осуществляется с помощью функций из библиотеки **kernel32.dll**. Вот некоторые из наиболее распространенных функций для работы с файлами в MASM32:

`CreateFile` – функция используется для создания или открытия файла. Она может открывать файлы для чтения, записи, или одновременно для чтения и записи.

`ReadFile` – функция используется для чтения данных из файла.

`WriteFile` – функция используется для записи данных в файл.

`SetFilePointer` – функция используется для перемещения указателя текущей позиции в файле.

`CloseHandle` – функция используется для закрытия файла после окончания работы с ним.

`GetFileSize` – функция используется для получения размера файла.

Функция **`CreateFile`** в MASM32 является системной функцией Windows API, которая используется для создания или открытия файла или устройства. Она принимает несколько параметров, включая имя файла или устройства, режим доступа, тип согласования доступа и другие параметры, определяющие поведение функции.

Прототип функции выглядит следующим образом:

```
invoke CreateFile,  addr lpFileName,
                    dwDesiredAccess,
                    dwShareMode,
                    addr lpSecurityAttributes,
                    dwCreationDisposition,
                    dwFlagsAndAttributes,
                    hTemplateFile
```

Параметры функции:

– **`lpFileName:`** указатель на строку, содержащую имя файла или устройства;

– **dwDesiredAccess:** режим доступа к файлу или устройству;

– **dwShareMode**: тип согласования доступа к файлу или устройству;

– **lpSecurityAttributes:** указатель на структуру SECURITY_ATTRIBUTES, которая определяет атрибуты безопасности для файла или устройства;

– **dwCreationDisposition:** тип создания файла или устройства;

– **dwFlagsAndAttributes:** дополнительные флаги и атрибуты для файла или устройства;

– **hTemplateFile:** дескриптор файла-шаблона.

Функция ***возвращает дескриптор открытого файла*** или устройства, который может использоваться в других функциях Windows API для работы с этим файлом или устройством. Если функция не может открыть файл или устройство, она возвращает значение INVALID_HANDLE_VALUE.

```
invoke CreateFile, ADDR filename, GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL
```

Здесь **filename** – это строка, содержащая имя файла;

GENERIC_READ – флаг доступа, указывающий, что файл должен быть открыт только для чтения;

FILE_SHARE_READ – флаг, указывающий, что другие процессы могут читать файл в то же время;

OPEN_EXISTING – флаг, указывающий, что файл должен быть открыт только если он уже существует;

FILE_ATTRIBUTE_NORMAL – атрибуты файла, указывающие, что файл должен быть обычным файлом без дополнительных атрибутов.

Функция **ReadFile** в MASM32 является системной функцией Windows API, которая используется для чтения данных из открытого файла или устройства. Она принимает несколько параметров, включая дескриптор файла, буфер для чтения, количество байтов для чтения и другие параметры, определяющие поведение функции.

Прототип функции выглядит следующим образом:

```
invoke ReadFile, hFile,
                 addr lpBuffer,
                 nNumberOfBytesToRead,
                 addr lpNumberOfBytesRead,
                 addr lpOverlapped
```

Параметры функции:

– **hFile**: дескриптор открытого файла или устройства, из которого нужно считать данные;

– **lpBuffer**: указатель на буфер, в который будут записаны считанные данные;

– **nNumberOfBytesToRead**: количество байтов, которые нужно прочитать из файла или устройства;

– **lpNumberOfBytesRead**: указатель на переменную, в которую будет записано количество фактически считанных байтов;

– **lpOverlapped**: указатель на структуру OVERLAPPED, которая используется для асинхронного чтения данных.

Функция возвращает значение TRUE, если операция чтения была успешно завершена, и FALSE в противном случае.

Пример использования функции **ReadFile**:

```
.data
   filename  db  "test.txt", 0
   buffer    db  100 dup(0)
   handle    dd  ?
   bytesread dd  ?

.code
invoke CreateFile, ADDR filename, GENERIC_READ, 0, NULL, OPEN_EXISTING,
\
                          FILE_ATTRIBUTE_NORMAL, NULL
      mov handle, eax              ; сохраняем дескриптор файла
      invoke ReadFile, handle, ADDR buffer, 100, ADDR bytesread, NULL
```

В этом примере функция **CreateFile** открывает файл "test.txt" для чтения и сохраняет дескриптор файла в переменную **handle**. Затем функция **ReadFile** читает до 100 байтов из файла в буфер **buffer** и сохраняет количество фактически считанных байтов в переменную **bytesread**.

Функция **WriteFile** в MASM32 является системной функцией Windows API, которая используется для записи данных в открытый файл или устройство. Она принимает несколько параметров, включая дескриптор файла, буфер с данными для записи, количество байтов для записи и другие параметры, определяющие поведение функции.

Прототип функции выглядит следующим образом:

```
invoke WriteFile, hFile,
                  addr lpBuffer,
                  nNumberOfBytesToWrite,
                  addr lpNumberOfBytesWritten,
                  addr lpOverlapped
```

Параметры функции:

– **hFile**: дескриптор открытого файла или устройства, в который нужно записать данные;

– **lpBuffer**: указатель на буфер с данными для записи;

– **nNumberOfBytesToWrite**: количество байтов, которые нужно записать в файл или устройство;

– **lpNumberOfBytesWritten**: указатель на переменную, в которую будет записано количество фактически записанных байтов;

– **lpOverlapped**: указатель на структуру OVERLAPPED, которая используется для асинхронной записи данных.

Функция возвращает значение TRUE, если операция записи была успешно завершена, и FALSE в противном случае.

```
.data
   filename   db   "test.txt", 0
   buffer     db   "Hello, world!", 0
   handle     dd   ?
   bytes      dd   ?

.code
     invoke CreateFile, ADDR filename, GENERIC_WRITE, 0, NULL, \
                        CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL
     mov handle, eax                   ; сохраняем дескриптор файла
     invoke WriteFile, handle, ADDR buffer, SIZEOF buffer, \
                       ADDR bytes, NULL
```

В этом примере функция **CreateFile** создает файл «test.txt» для записи, а функция **WriteFile** записывает строку «Hello, world!» в файл. Функция sizeof используется для определения размера буфера в байтах.

Обратите внимание, что при записи данных в файл с помощью функции **WriteFile** не происходит автоматической конвертации данных в текстовый формат. Если вы хотите записать данные в файл в виде текста, вам нужно преобразовать их в строку ASCII или Unicode перед вызовом функции **WriteFile**.

Размещение элементов массива в файле зависит от того, какой формат вы выбираете для записи массива в файл.

Если вы хотите записать массив в бинарном формате, то каждый элемент массива будет занимать фиксированное количество байтов (например, один байт для массива типа *byte*, два байта для массива типа *word*, четыре байта для массива типа *dword* и т. д.). В этом случае элементы массива должны быть записаны в файл подряд, без разделителей или других метаданных.

Например, если вы хотите записать массив типа *byte* в файл, то каждый элемент массива будет занимать один байт, и вы можете записать массив в файл следующим образом:

```
42 17 23 10 55 33 12 64 99 27
```

Если вы хотите записать массив в текстовом формате, то каждый элемент массива будет записан как строка символов, разделенных пробелами, запятыми или другими разделителями. В этом случае элементы массива должны быть записаны в файл с использованием символьной кодировки, например, ASCII или UTF-8.

Так, если вы хотите записать массив типа *byte* в текстовый файл, то каждый элемент массива может быть записан как двузначное шестнадцатеричное число, разделенное пробелами:

```
2A 11 17 0A 37 21 0C 40 63 1B
```

При чтении массива из файла в программе вы должны учитывать формат записи массива в файле и правильно интерпретировать данные, чтобы получить правильный массив в памяти.

Лабораторная работа № 8

Цель работы:

– научиться создавать библиотечные функции;

– освоить операции работы с файлами.

Задания для лабораторной работы

1. Составить блок-схему алгоритма и написать текст функции в соответствии с индивидуальным заданием.

2. Оформить функцию в виде библиотечного файла.

3. Написать програму для сохранения цифрового массива в бинарный файл, например, **array.hex**. При этом цифровой массив не следует конвертировать в текстовую строку.

4. Открыть созданный бинарный файл в редакторе, например **notepad++**. Для этого в редакторе notepad++ потребуется установить дополнение **HEX-Editor**. Это можно сделать в меню *Плагины –> Управление плагинами…* (рис. 8.1 и 8.2).

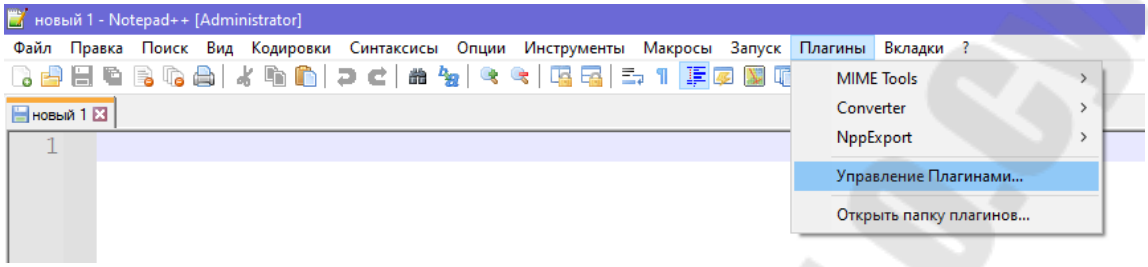


Рис. 8.1. Подключение плагина в Notepad++

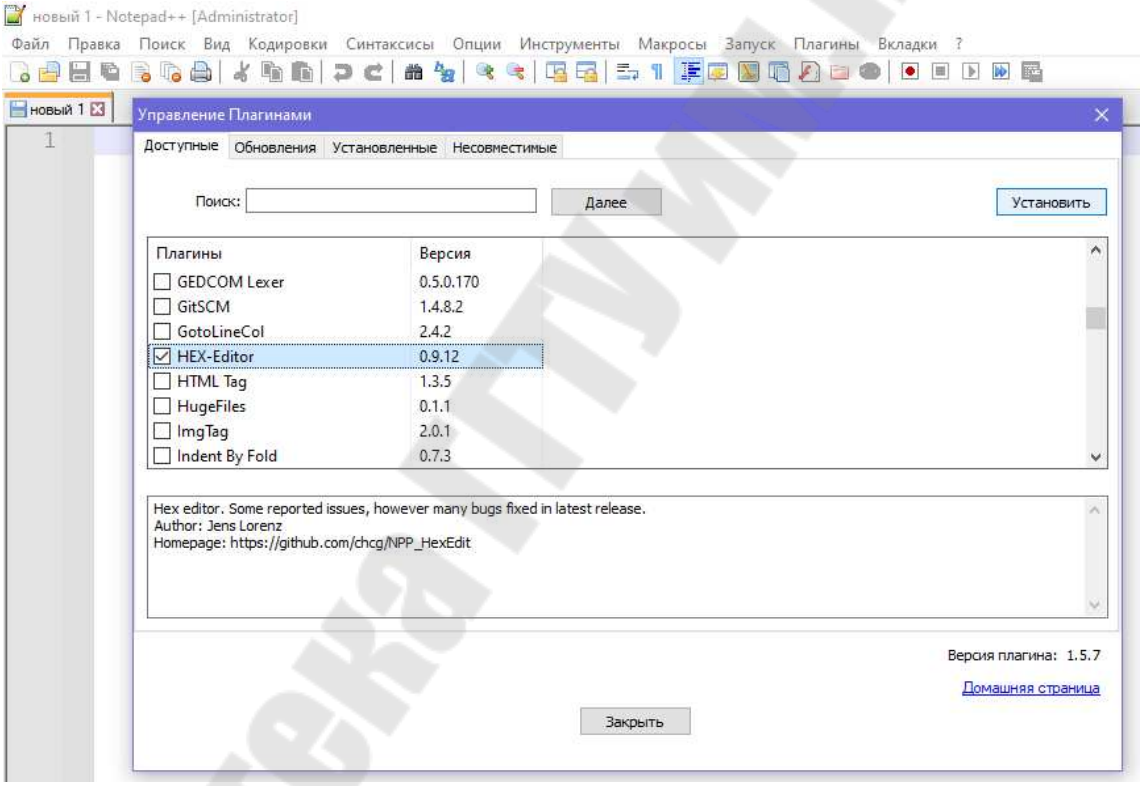


Рис. 8.2. Выбор плагина HEX-Editor в Notepad++

Для открытия бинарного файла в дополнении HEX-Editor необходимо выбрать меню *Плагины –> HEX-Editor –> View in HEX*.

Пример цифрового массива из десяти цифр формата DWORD (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) в *HEX-Editor* (рис. 8.3).

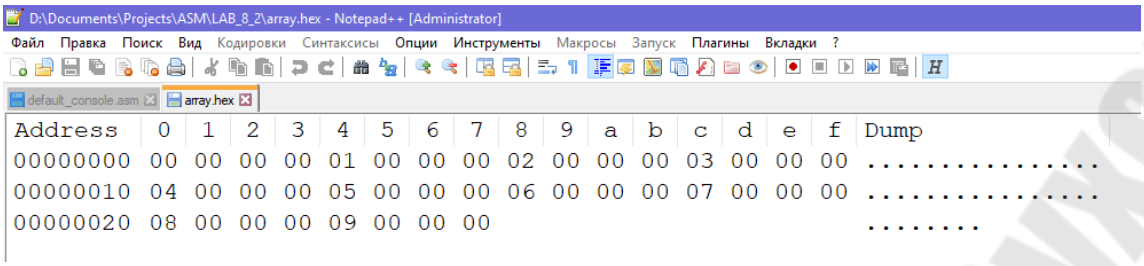


Рис. 8.3. Представление цифрового массива в *HEX-Editor Notepad++*

5. Изменить значения элементов массива в HEX-редакторе, и сохранить бинарный файл под новым имененем.

6. Написать программу, которая выполнит чтение нового HEX-файла, произведет обработку измененного массива с использованием полученной библиотечной функции и сохранит результат в текстовый файл, например, **result.txt**.

7. Разработать и выполнить все необходимые тесты программы. Для каждого задания необходимо создать не менее семи переменных типа заданной структуры и выполнить над ними необходимое действие.

8. Индивидуальные задания, в соответствии с номером студента в журнале группы, приведены в табл. 8.1.

Таблица 8.1

Задания к лабораторной работе № 8

Вариант	Задание
01	Найти минимальный элемент в каждой строке двумерного массива
02	Посчитать сумму элементов в каждом столбце двумерного массива
03	Проверить, является ли каждый элемент двумерного массива четным
04	Найти количество четных элементов в двумерном массиве
05	Поменять знак у всех элементов двумерного массива
06	Проверить, является ли каждый элемент двумерного массива положительным
07	Найти наименьший положительный элемент в двумерном массиве
08	Проверить, является ли каждый элемент двумерного массива простым числом
09	Найти количество простых чисел в двумерном массиве
10	Найти сумму элементов двумерного массива, которые делятся на 3
11	Найти все индексы заданного элемента в двумерном массиве

Окончание табл. 8.1

Вариант	Задание
12	Найти индексы первого вхождения заданного элемента в двумерном массиве
13	Найти произведение элементов двумерного массива, которые делятся на 2
14	Найти наибольший отрицательный элемент в двумерном массиве
15	Найти максимальный из элементов двумерного массива, которые расположены на четных строках и столбцах
16	Проверить, является ли каждый элемент двумерного массива уникальным
17	Найти наименьший неуникальный элемент в двумерном массиве
18	Проверить, является ли двумерный массив квадратным (все строки и столбцы имеют одинаковую длину)
19	Найти максимальный элемент в каждой строке двумерного массива
20	Найти максимальный из элементов двумерного массива, расположенных на главной диагонали
21	Найти максимальный элемент в двумерном массиве и вернуть его координаты
22	Проверить, является ли двумерный массив симметричным относительно своей побочной диагонали
23	Проверить, является ли двумерный массив симметричным относительно своей главной диагонали
24	Найти сумму элементов двумерного массива, которые находятся ниже побочной диагонали
25	Найти сумму элементов двумерного массива, которые находятся ниже главной диагонали
26	Поменять местами столбцы двумерного массива по заданным индексам
27	Найти сумму элементов двумерного массива, которые находятся на побочной диагонали
28	Проверить, является ли первый столбец двумерного массива упорядоченным по убыванию
29	Проверить, является ли вторая строка двумерного массива упорядоченной по убыванию
30	Найти сумму максимальных элементов каждой строки двумерного массива

Содержание отчета:

1. Название работы.
2. Цель работы.
3. Постановка задачи.
4. Блок-схема алгоритма.
5. Текст программ с комментариями.
6. Скриншоты содержимого всех созданных программами файлов и тестов программы.
7. Выводы.

Контрольные вопросы

1. Как подключить библиотечные функции в MASM32?
2. Каким образом осуществляется вызов функции из библиотеки в MASM32?
3. В чем отличие DLL с точкой входа и без точки входа?
4. Что такое динамическое подключение библиотек?
5. Каким образом можно открыть файл в MASM32?
6. Какие функции используются для чтения данных из файла в MASM32?
7. Каким образом можно записать данные в файл в MASM32?
8. Как закрыть файл в MASM32?
9. Как размещаются элементы массива в файле?
10. Что такое HEX-файл, как просмотреть его содержимое?

Глава 9. Знакомство с работой в среде программирования микроконтроллеров AVR Studio

Среда разработки программ AVR Studio

Разработка программ в AVR Studio начинается с создания проекта. После установки программы это легче всего сделать через менеджер создания проектов во вкладке Project/Project Wizard.

После нажатия экранной кнопки New Project появится окно, приведенное на рис. 9.1, в котором надо задать название и директорию размещения проекта, например, 168_LED (168, потому что для Atmega168, а LED, потому что программа будет управлять светодиодами). В качестве типа проекта необходимо выбрать Atmel AVR Assembler (проекты AVR GCC используют Си-компилятор WinAVR). Сейчас и в дальнейшем очень важно размещать все файлы проекта в одной папке, что избавит от многих проблем при редактировании и переносе программ.

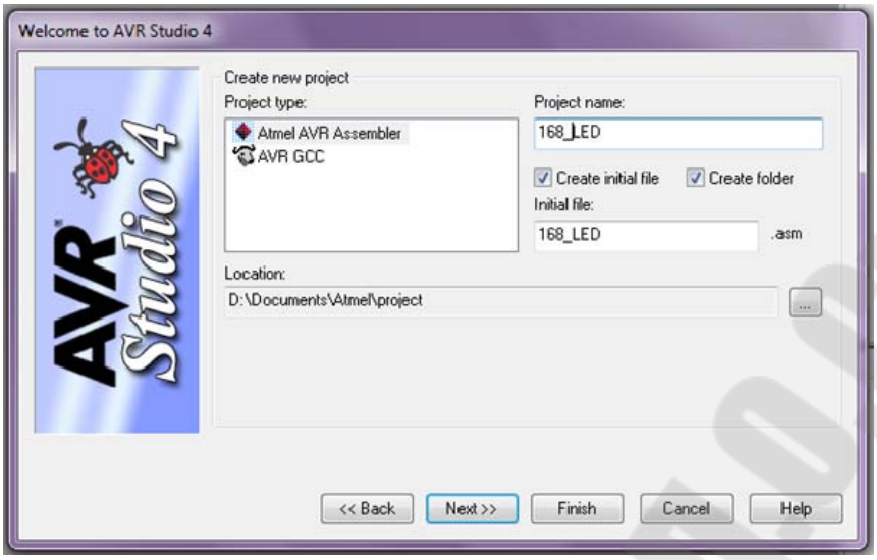


Рис. 9.1. Окно Project Wizard AVR Studio

ВНИМАНИЕ! Путь к программе и папке с сохраненными проектами не должен содержать имен, написанных кириллицей.

На следующем этапе необходимо выбрать модель микроконтроллера и тип отладочного средства, как показано на рис. 9.2 (в нашем случае это ATmega168 и AVR Simulator соответственно), после чего откроется окно текстового редактора, где и будет непосредственно происходить создание программы (рис. 9.3).

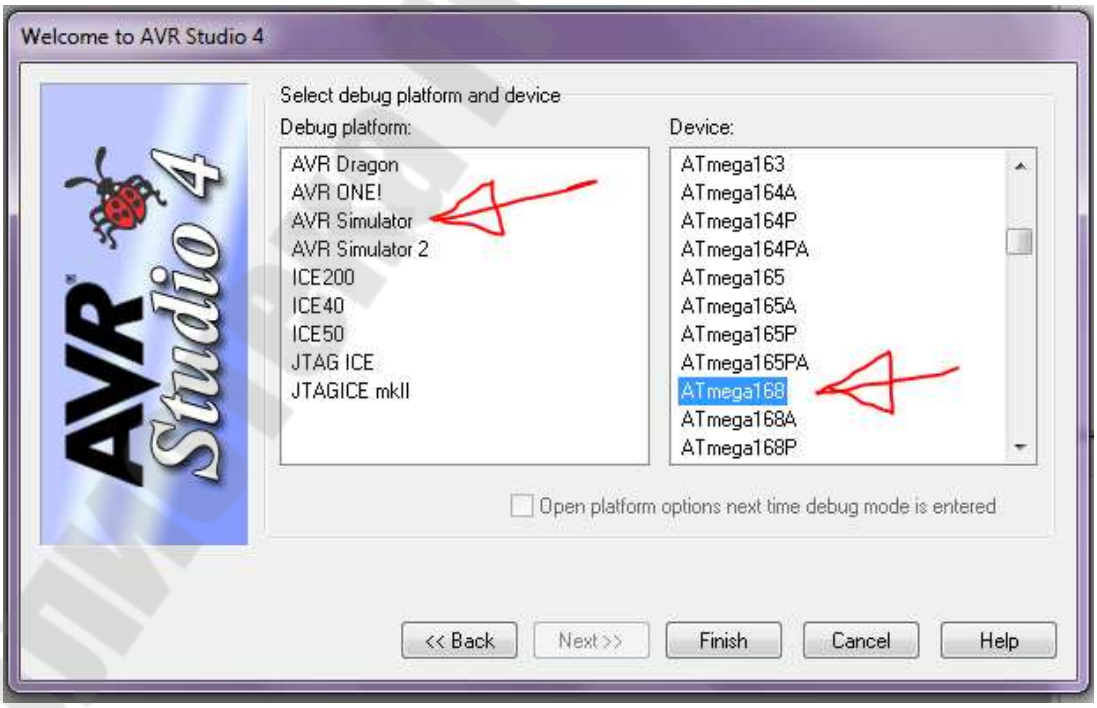


Рис. 9.2. Выбор отладчика (debug platform) и модели микроконтроллера

В окне проекта Project можно видеть все компоненты программы: файл с исходным текстом 168_LED_a.asm, файл описания контроллера m168def.inc, а также выходные файлы с расширениями .map, .hex и .obj с одноименными именами. В разделе Labels находятся символьные имена меток, встречающиеся в программе.

Компиляция проекта осуществляется после нажатия на иконку Assemble либо Assemble and Run. В последнем случае сразу же запускается и программа отладчика. Если в исходном тексте были допущены ошибки, то .hex, естественно, создан не будет, а в окне Build появится описание всех ошибок и строки, где они находятся. После внесения необходимых исправлений и успешной сборки в окне Build отобразится статистика о проекте в виде диапазонов адресов и размеров секций FLASH, SRAM и EEPROM.

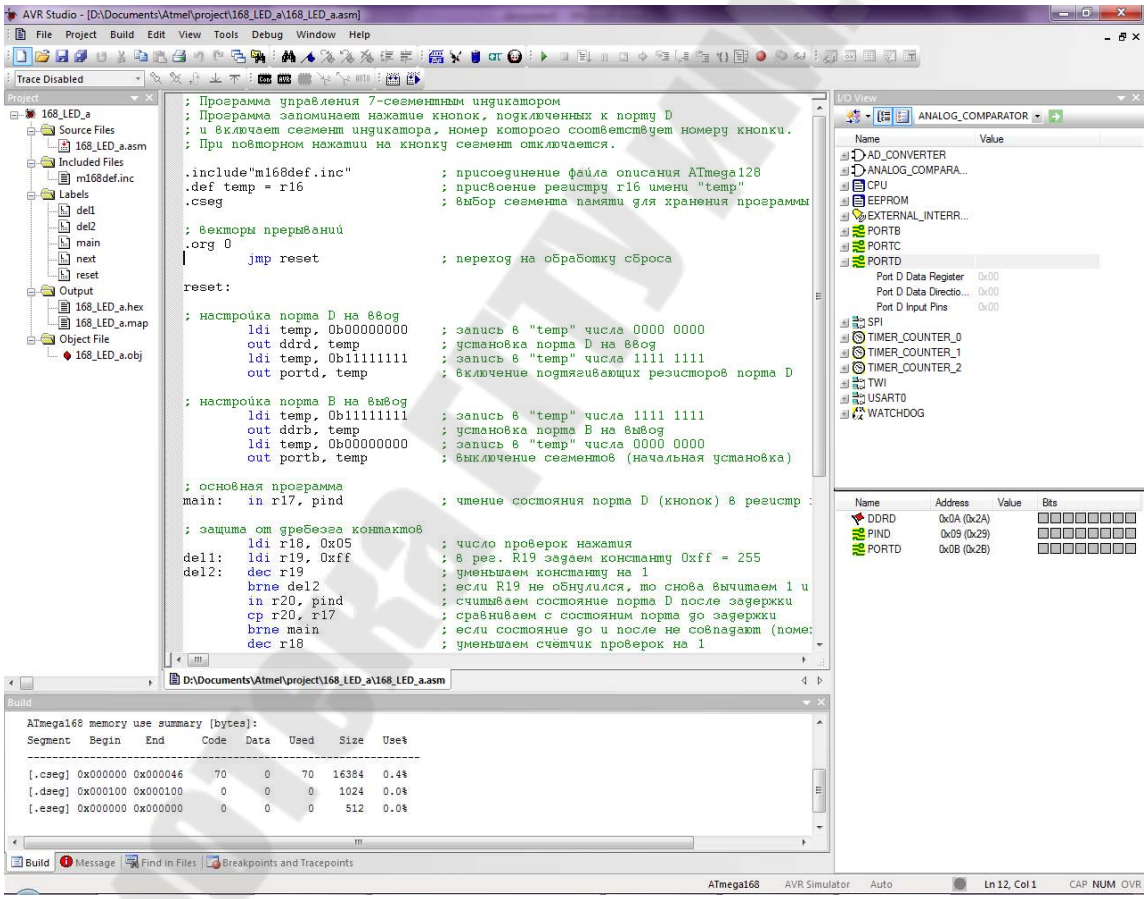


Рис. 9.3. Текстовый редактор AVR Studio

Проверить работоспособность программы можно в симуляторе либо с помощью любого другого отладчика. Его запуск происходит после нажатия иконки Start Debagging (иконка в виде зеленого треугольника на панели инструментов).

На рис. 9.4 показана работа в симуляторе и вид основных отладочных окон, в которых можно наблюдать за состоянием содержимого различных областей памяти и символьными именами, объявленных в тексте программы. Все эти данные доступны для редактирования на ходу.

Отладку можно вести как в пошаговом (кнопки Step Into, Step Over, Step Out), так в автоматическом (Auto Step) или ускоренном (Run) режимах. Имеется возможность использовать также ***точки останова***. Симулятор, встречая строку, в которой находится точка останова, принудительно останавливает свое выполнение, после чего можно детально изучить содержимое отладочных окон. Управление точками останова производится кнопками Toggle Brekpoint (включение точки останова) и Remove all Program Brekpoints (удаление всех точек останова). В окне Disassembler можно видеть соответствие машинных кодов командам Ассемблера AVR.

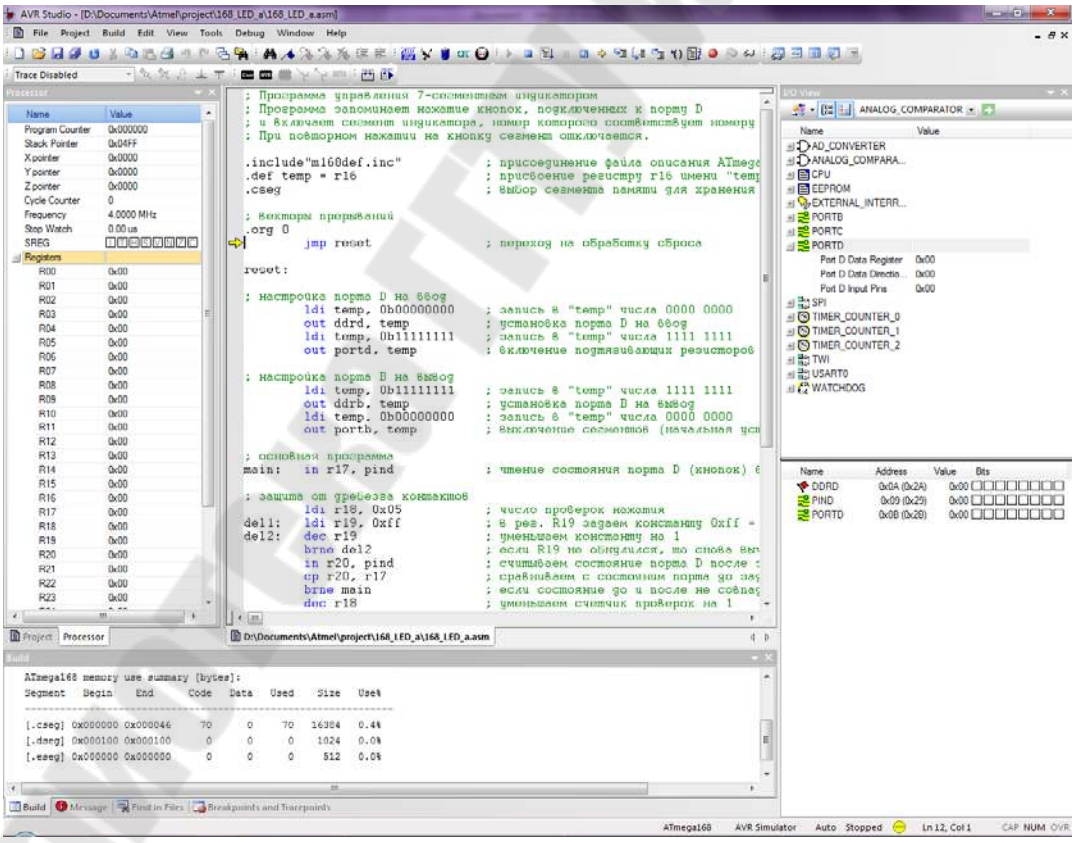


Рис. 9.4. Работа в симуляторе AVR Studio

Детальное описание всех компонентов AVR Studio можно найти во встроенной справочной системе.

Порты микроконтроллера

С внешним миром микроконтроллер общается через порты ввода/вывода. Каждый порт микроконтроллера AVR (обычно имеют имена *A*, *B* и т. д.) имеет 8 разрядов, каждый из которых привязан к определенной ножке (выводу, пину) корпуса. Каждый порт имеет три специальных регистра DDR*x*, PORT*x* и PIN*x* (где *x* соответствует букве порта *A*, *B*, *C* и т. д.). Назначение регистров следующее.

PIN*x* – это регистр чтения. Из него можно только читать. В регистре PIN*x* содержится информация о реальном текущем логическом уровне на выводах порта. Вне зависимости от настроек порта. Так что, если хотим узнать что у нас на входе, – читаем соответствующий бит регистра PIN*x*.

DDR*x* – это регистр направления порта. Отдельные разряды порта в конкретный момент времени могут быть либо входом, либо выходом (но для состояния битов PIN*x*, это значения не имеет. Читать из PIN*x* реальное значение можно всегда).

Если DDRx_n = 0 – *n*-й разряд порта *x* работает как ВХОД (ввод данных).

Если DDRx_n = 1 – *n*-й разряд порта *x* работает как ВЫХОД (вывод данных).

PORT*x* – режим управления состоянием разрядов порта. Когда мы настраиваем какие-либо разряды порта на ввод данных, то от PORT*x* зависит тип входа (Hi-Z или PullUp).

Когда *n*-й разряд порта настроен на ввод, и при этом PORTx_n = 0, то данный разряд находится в режиме Hi-Z (без подтягивающего резистора).

Если PORTx_n = 1, то *n*-й разряд порта переходит в режим *PullUp* – внутри микроконтроллера включается подтягивающий резистор величиной 100 кОм между выводом порта и шиной питания.

Когда *n*-й разряд порта настроен как выход, то значение соответствующего бита в регистре PORT*x* определяет значение этого разряда порта. Если PORTx_n = 1, то на выводе логическая «1», если PORTx_n = 0, то на выводе логический «0».

Лабораторная работа № 9

Цель работы:

– изучить особенности работы со средой программирования микроконтроллеров AVR Studio;

– ознакомиться с работой портов микроконтроллера;

– получить навыки написания простейшей программы на языке Ассемблера;

– исследовать работу простейшей программы с помощью встроенного в AVR Studio эмулятора работы микроконтроллера.

Задания к лабораторной работе

В окне текстового редактора AVR Studio ввести текст программы управления светодиодами. Восемь светодиодов подключены к порту B микроконтроллера Atmega168. Восемь нормально разомкнутых контактов подключены к порту D. Светодиоды должны светиться при нажатии на соответствующий контакт.

Текст программы:

```
.include "m168def.inc"          ; подключаем файл описания
.def  temp = r16                ; присваиваем имя регистру R16
.cseg                           ; указываем сегмент памяти программ
.org 0                          ; начальный адрес программы
rjmp reset                      ; переход к метке «reset»

reset:
ldi temp, 0b00000000            ; настраиваем порт D на ввод и
out ddrd, temp                  ; подключаем подтягивающие резисторы
ldi temp, 0b11111111
out portd, temp

ldi temp, 0b11111111            ; настраиваем порт B на вывод
out ddrb, temp                  ; и выключаем сегменты
ldi temp, 0b00000000
out portb, temp

main:                           ; основная программа
in temp, pind                   ; читаем состояние порта D
com temp                        ; инвертируем полученные значения
out portb, temp                 ; записываем результат в порт B
rjmp main                       ; переходим к началу основной прогр.
```

Привести комментарии к командам и директивам программы.

Произвести отладку программы.

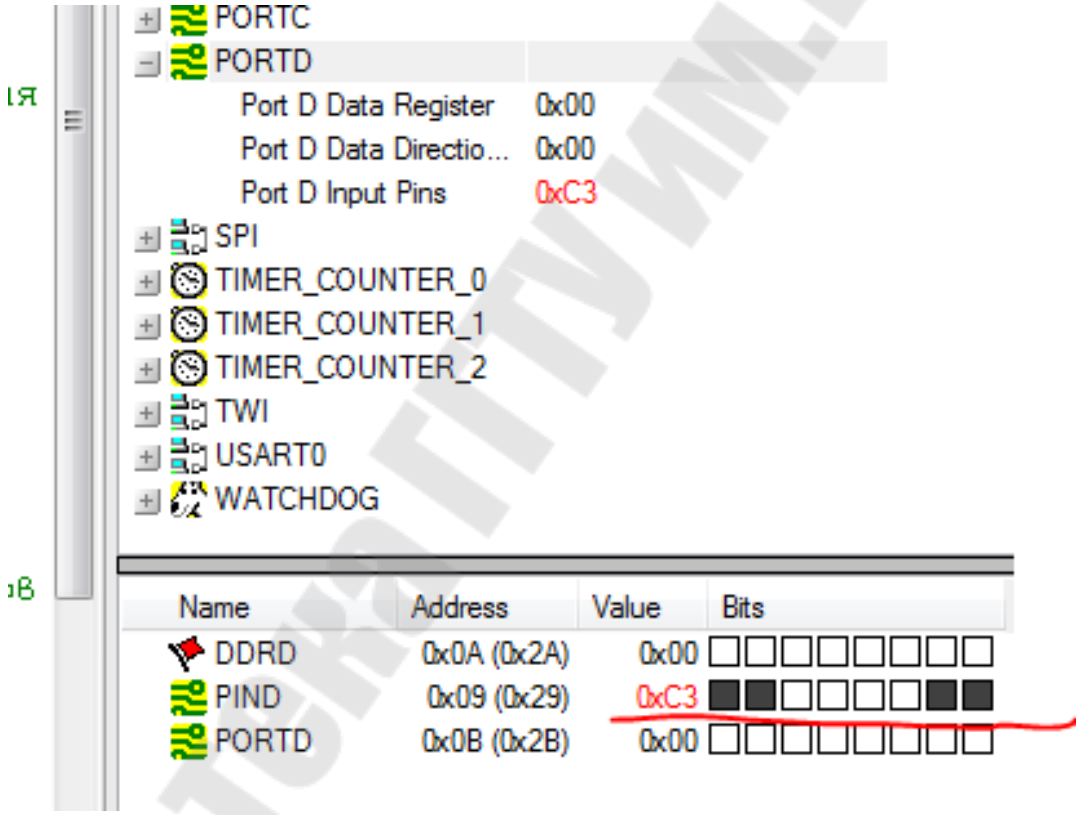
Выполняя программу в пошаговом режиме, после выполнения каждого шага, записывать в табл. 9.1 содержимое регистров, занятых в выполнении программы.

Таблица 9.1

Состояние регистров контроллера

Шаг №	PC	temp	DDRD	PORTD	PIND	DDRB	PORTB	PINB	Комментарии
0	0	0	0	0	0	0	0		Исходные значения
1									
…									

ПРИМЕЧАНИЕ! Перед выполнением команды **`in temp, pind`** необходимо вручную установить произвольное значение разрядов регистра PIND, кликнув по пустым битам левой кнопкой мыши, например, так:



Содержание отчета:

1. Наименование и цель работы.
2. Текст программы с комментариями к командам и директивам.
3. Заполненная в соответствии с заданием табл. 9.1 с подробными комментариями.
4. Сделать выводы по существу полученных результатов.

Контрольные вопросы

1. Что представляет собой среда разработки AVR Studio?
2. Какие основные типы регистров присутствуют в микроконтроллерах ATMega?
3. Каким образом можно прочитать значение регистра в AVR Studio?
4. Как происходит запись значения в регистр в AVR Studio?
5. Какой регистр используется для управления режимом работы портов ввода-вывода?
6. Каким образом можно установить флаги состояния в AVR Studio?
7. Как можно использовать регистры общего назначения для временного хранения данных в AVR Studio?
8. Что такое порты ввода-вывода микроконтроллера?

Литература

1. Юров, В. И. Assembler : учебное пособие для вузов / В. И. Юров. – 2-е изд. – Санкт-Петербург : Питер, 2007. – 636 с.
2. Юров, В. И. Assembler : практикум / В. И. Юров. – 2-е изд. – Санкт-Петербург : Питер, 2007. – 398 с.
3. Лисицин, Д. В. Программирование на языке Ассемблера : учебное пособие / Д. В. Лисицин ; Новосибирский государственный технический университет. – Новосибирск : Новосиб. гос. техн. ун-т, 2018. – 100 с. : ил., табл. – URL: https://biblioclub.ru/index.php?page=book&id=574827 (дата обращения: 25.03.2024).
4. Секаев, В. Г. Основы программирования на Ассемблере : учебное пособие / В. Г. Секаев. – Новосибирск : Новосиб. гос. техн. ун-т, 2010. – 100 с. – URL: https://biblioclub.ru/index.php?page=book&id=228986 (дата обращения: 25.03.2024).
5. Основы программирования микропроцессоров Intel для встраиваемых систем : учебное пособие / С. В. Скороход, В. В. Селянкин, С. Н. Дроздов [и др.] ; Южный федеральный университет, Инженерно-технологическая академия. – Таганрог : Юж. федер. ун-т, 2016. – 82 с. : табл. – URL: https://biblioclub.ru/index.php?page=book&id=493316 (дата обращения: 25.03.2024).
6. Пьявченко, А. О. Архитектура, основы программирования и применения AVR-микроконтроллеров и ARM-микросистем : учебное пособие / А. О. Пьявченко, В. А. Переверзев ; Южный федеральный университет. – Ростов-на-Дону ; Таганрог : Юж. федер. ун-т, 2019. – Ч. 1. – 376 с. : ил. – URL: https://biblioclub.ru/index.php?page=book&id=598674 (дата обращения: 25.03.2024).
7. Абрамов, Е. С. Машинно-ориентированное программирование : учебное пособие / Е. С. Абрамов, И. Д. Сидоров. – Таганрог : Южный федеральный университет, 2016. – 88 с. : схемы, табл., ил. – URL: https://biblioclub.ru/index.php?page=book&id=492941 (дата обращения: 25.03.2024).
8. Кирнос, В. Н. Введение в вычислительную технику : основы организации ЭВМ и программирование на Ассемблере : учебное пособие / В. Н. Кирнос ; Томский государственный университет систем управления и радиоэлектроники. – Томск : Эль Контент, 2011. – 172 с. : ил., табл., схемы. – URL: https://biblioclub.ru/index.php?page=book&id=208652 (дата обращения: 25.03.2024).

9. Пильщиков, В. Н. Программирование на языке Ассемблера IBM PC : учебное пособие / В. Н. Пильщиков. – Москва : Диалог-МИФИ, 2014. – 288 с. : ил. – URL: https://biblioclub.ru/index.php?page=book&id=447687 (дата обращения: 25.03.2024).

Учебное электронное издание комбинированного распространения

Учебное издание

Савельев Вадим Алексеевич
Дорощенко Игорь Васильевич

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕРА

Учебно-методическое пособие

Электронный аналог печатного издания

Редактор *Н. Г. Мансурова*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 28.02.25.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Цифровая печать. Усл. печ. л. 7,21. Уч.-изд. л. 7.
Изд. № 12.
http://www.gstu.by

Издатель и полиграфическое исполнение
Гомельский государственный
технический университет имени П. О. Сухого.
Свидетельство о гос. регистрации в качестве издателя
печатных изданий за № 1/273 от 04.04.2014 г.
пр. Октября, 48, 246746, г. Гомель