Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО Кубанский государственный технологический
университет

Кафедра информационных систем и программирования

НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

Методические указания к практическим занятиям для студентов всех
форм обучения направления подготовки
09.03.04 Программная инженерия

Краснодар
2022

Составители: канд. техн. наук, доц. А. Г. Мурлин;
ст. преп. А. Г. Волик;
ст. преп. Е. А. Симоненко

Низкоуровневое программирование. Методические указания к практическим занятиям для студентов всех форм обучения направления подготовки бакалавров 09.03.04 Программная инженерия / Сост.: А. Г. Мурлин, А. Г. Волик, Е. А. Симоненко; Кубан. гос. технол. ун-т. Каф. информационных систем и программирования. – Краснодар. 2022 – 135 с.

Содержат описания практических работ, указания к их выполнению, задания и требования к оформлению отчёта. Изложены основы разработки программ на языке ассемблера и принципы работы микропроцессора. Рассмотрены пользовательские регистры и основные команды микропроцессоров семейства x86, показаны примеры создания и процесс исполнения процедур и обработчиков прерываний, описана работа со строковыми командами, а также средствами ввода-вывода. Приведены исходные тексты примеров программ и рекомендуемая литература.

Ил.7. Табл.3. Библиогр.: 6 назв.

Рецензенты: Зав. кафедрой ИСП, канд. техн. наук, доц.
М.В. Янаева;
руководитель отдела телекоммуникаций Краснодарского регионального информационного центра сети «Консультант-Плюс», канд. техн. наук
Н.Ф. Григорьев

Содержание

Введение 4
Практическое занятие № 1. Разработка программ на языке ассемблера в интегрированной среде Microsoft Visual Studio. Структура программы на языке ассемблера 6
Практическое занятие № 2. Регистры. Организация памяти и режимы адресации. Команды пересылки данных 33
Практическое занятие № 3. Арифметические команды целочисленного устройства микропроцессора 49
Практическое занятие № 4. Двоично-десятичная арифметика 67
Практическое занятие № 5. Команды логических операций 72
Практическое занятие № 6. Команды передачи управления 77
Практическое занятие № 7. Команды сдвигов 96
Практическое занятие № 8. Подпрограммы и вызов процедур 102
Практическое занятие № 9. Цепочечные команды 113
Список рекомендуемой литературы 121
Приложение А (справочное) Организация памяти 122
Приложение Б (справочное) Содержимое регистра флагов 124
Приложение В (справочное) Команды безусловного перехода 128

Введение

Методические указания содержат описание практических занятий по дисциплине «Микропроцессорные системы» для направления подготовки бакалавров 09.03.03 Прикладная информатика.

Целью выполнения практических занятий является закрепление основ и углубление знаний в области микропроцессорной техники и языка ассемблера, изучение системы команд микропроцессоров семейства x86 (архитектуры IA-32), а также ознакомление студентов со средствами компиляции и отладки программ и принципах написания низкоуровневых программ. В практических занятиях рассмотрены как 16-ти, так и 32-х битные инструкции и режимы адресации, основные флаги и пользовательские регистры. Показаны примеры создания и процесс исполнения процедур и обработчиков прерываний, описана работа с математическим сопроцессором, строковыми командами, а также средствами ввода-вывода, как текстового, так и графического. Даны примеры разработки программ для ОС Windows.

Ассемблер – это самый старый из языков программирования и, в отличие от всех остальных языков, он тесно связан с архитектурой процессора. Он является языком низкого уровня, на котором программист работает непосредственно с инструкциями микропроцессора. С его помощью программист получает прямой доступ к аппаратным ресурсам компьютера. Для программирования на ассемблере требуются глубокие познания в архитектуре компьютера и структуре операционной системы. Понимание основ архитектуры современных ПК и применение этих знаний при написании программ позволяют улучшить скорость выполнения и эффективность использования памяти, даже если непосредственно не производится использование ассемблерных команд. Поэтому дисциплина «Микропроцессорные системы» является важным этапом подготовки инженеров по направлению подготовки бакалавров 09.03.03 Прикладная информатика.

При выполнении практических занятий должен соблюдаться следующий порядок выполнения работы:

- ознакомиться с описанием практической работы;
- получить номер варианта задания у преподавателя;
- изучить необходимый теоретический материал, пользуясь настоящими указаниями и рекомендованной литературой;
- написать программу в соответствии с вариантом задания и отладить ее на ЭВМ;
- подготовиться к ответам на теоретические вопросы по теме практической работы;
- оформить отчет.

Все студенты должны предъявить индивидуальный отчет о результатах выполнения практической работы. Допускается предъявление отчета в виде электронного документа.

Отчет должен содержать следующие пункты:

1) Титульный лист.
2) Наименование и цель работы.
3) Краткое теоретическое описание (основные моменты).
4) Задание на практическую работу, включающее четкую формулировку задачи.
5) Результаты выполнения работы.
6) Листинг программы.

При сдаче отчёта студент должен показать знание теоретического материала в объёма, определяемом тематикой практической работы, а также пониманием сущности выполняемой работы.

Практическое занятие № 1. Разработка программ на языке ассемблера в интегрированной среде Microsoft Visual Studio. Структура программы на языке ассемблера

1 Цель работы

Изучить инструментальные средства разработки и возможности отладки программ на языке ассемблера в интегрированной среде Microsoft Visual Studio.

2 Краткая теория

Среда разработки Microsoft Visual Studio – это набор инструментов и средств, предназначенных для разработчиков программ, с широким набором поддерживаемых языков программирования. Интегрированная среда разработки (Integrated Development Environment, IDE) Visual Studio поддерживает множество языков программирования, например C#, C++, F# и ряд других. Однако язык ассемблера не поддерживается изначально и не входит в состав Visual Studio.

Одним из решений является использование дополнительных настроек среды для поддержки работы с языком ассемблера. Для этого необходимо выполнить ряд действий для установки и настройки.

> *Примечание*. Перед началом работы, для повышения удобства использования можно установить расширение Assembly Language Syntax Highlighting для Visual Studio, добавляющее поддержку подсветки синтаксиса языка ассемблера. Его можно скачать с сайта проекта на CodePlex (http://asmhighlighter.codeplex.com/).
> При возникновении проблем с подсветкой синтаксиса можно попробовать удалить ветку реестра:

```
HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\11.0\FontAndColors\Cache
```

2.1 Настройка и создание проекта

2.1.1 Создание проекта

Для начала работы необходимо создать новый пустой проект.

Для создания проекта нужно выбрать соответствующий пункт меню среды разработки (File -> New -> Project) или нажать комбинацию клавиш <Ctrl+Shift+N>. При этом появится диалоговое окно New Project, позволяющее создать все типы проектов Visual Studio (рисунок 1.1).

С начала необходимо выбрать тип проекта. В данном случае нам необходим шаблон из группы приложений Visual C++ - Empty Project (Пустой проект). После этого вводим имя приложения и решения (Solution) (например, MPSLab101 и MPSLabs, соответственно). Далее нажимаем «OK» и попадаем в основное окно среды разработки (рисунок 1.2).

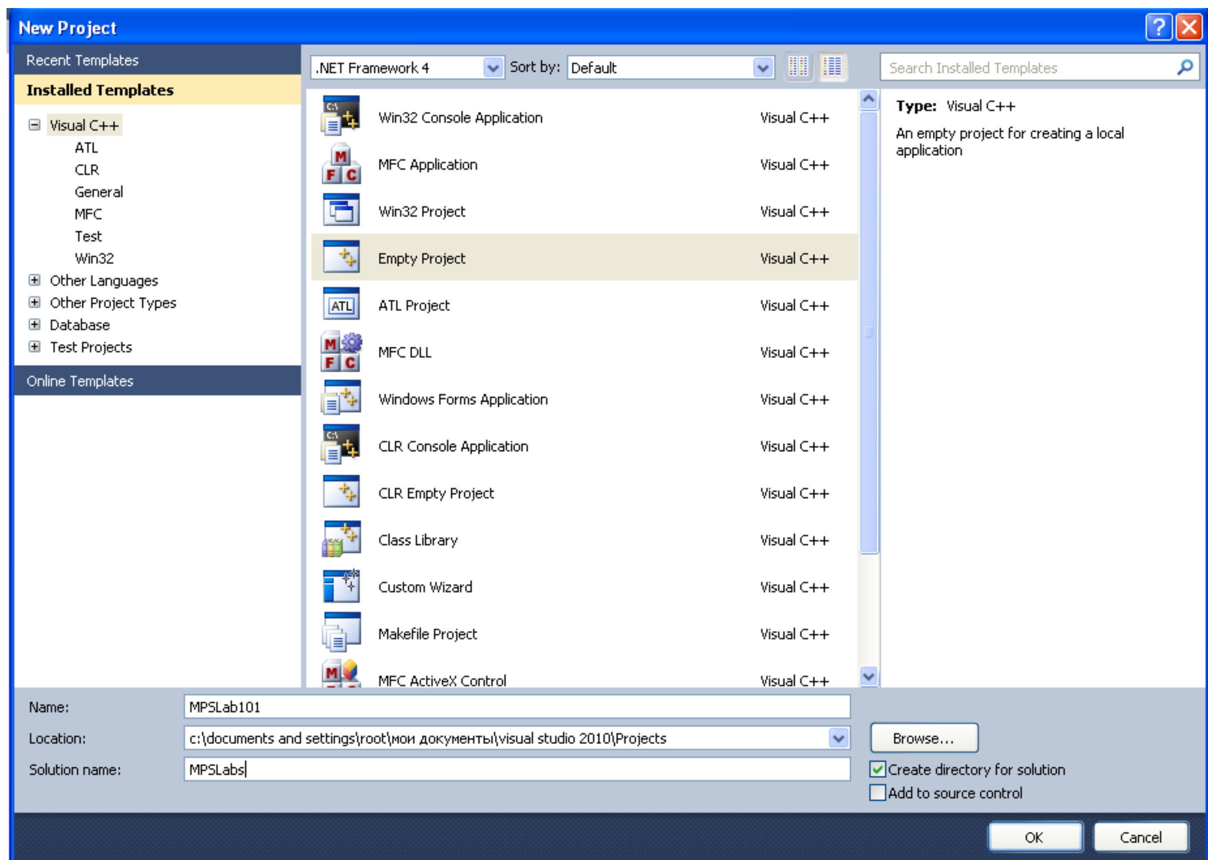


Рисунок 1.1 – Окно New Project

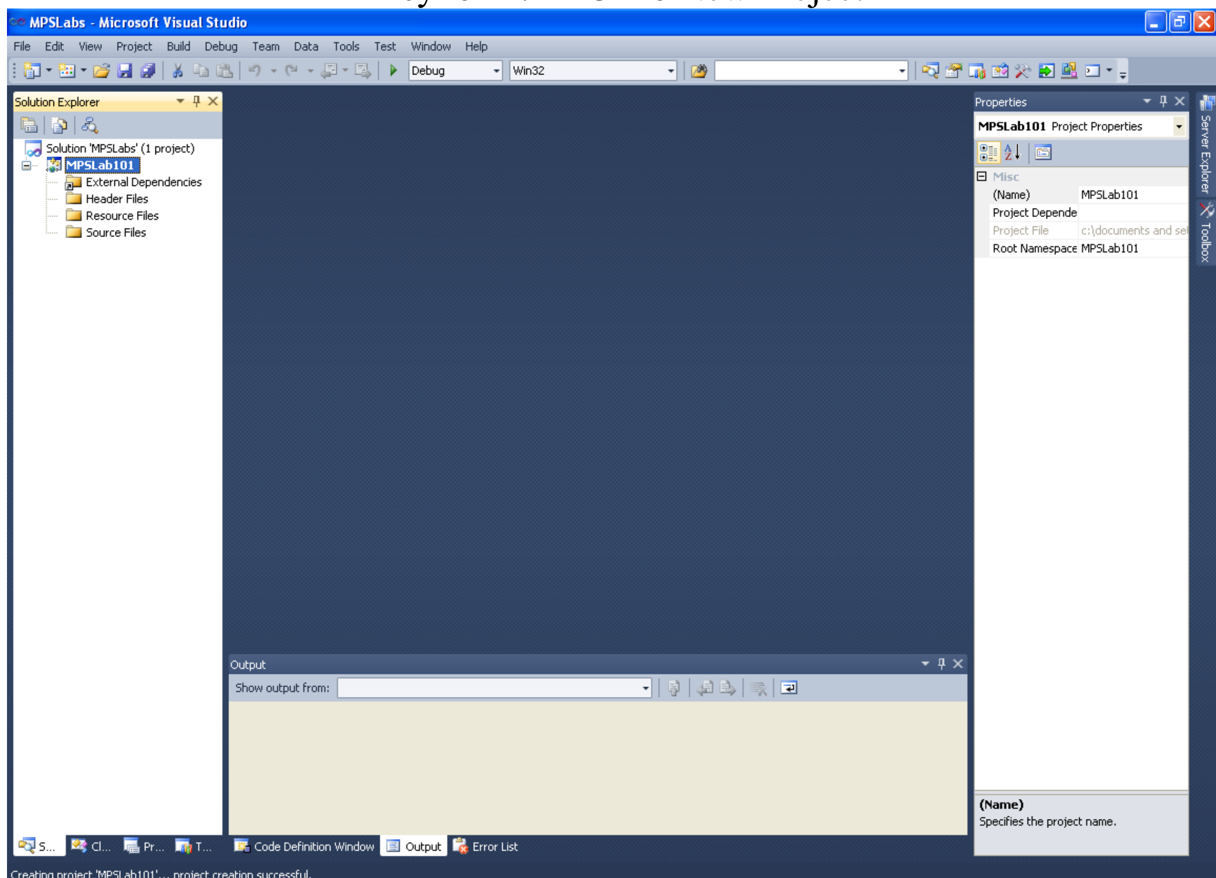


Рисунок 1.2 – Основное окно среды разработки

2.1.2 Добавление поддержки компилятора MASM

Далее необходимо нажать правой кнопкой мыши над именем проекта и выбрать меню «Build Customizations…» (рисунок 1.3).

В открывшемся окне необходимо выбрать пункт «masm (.targets, .props)» (рисунок 1.4).

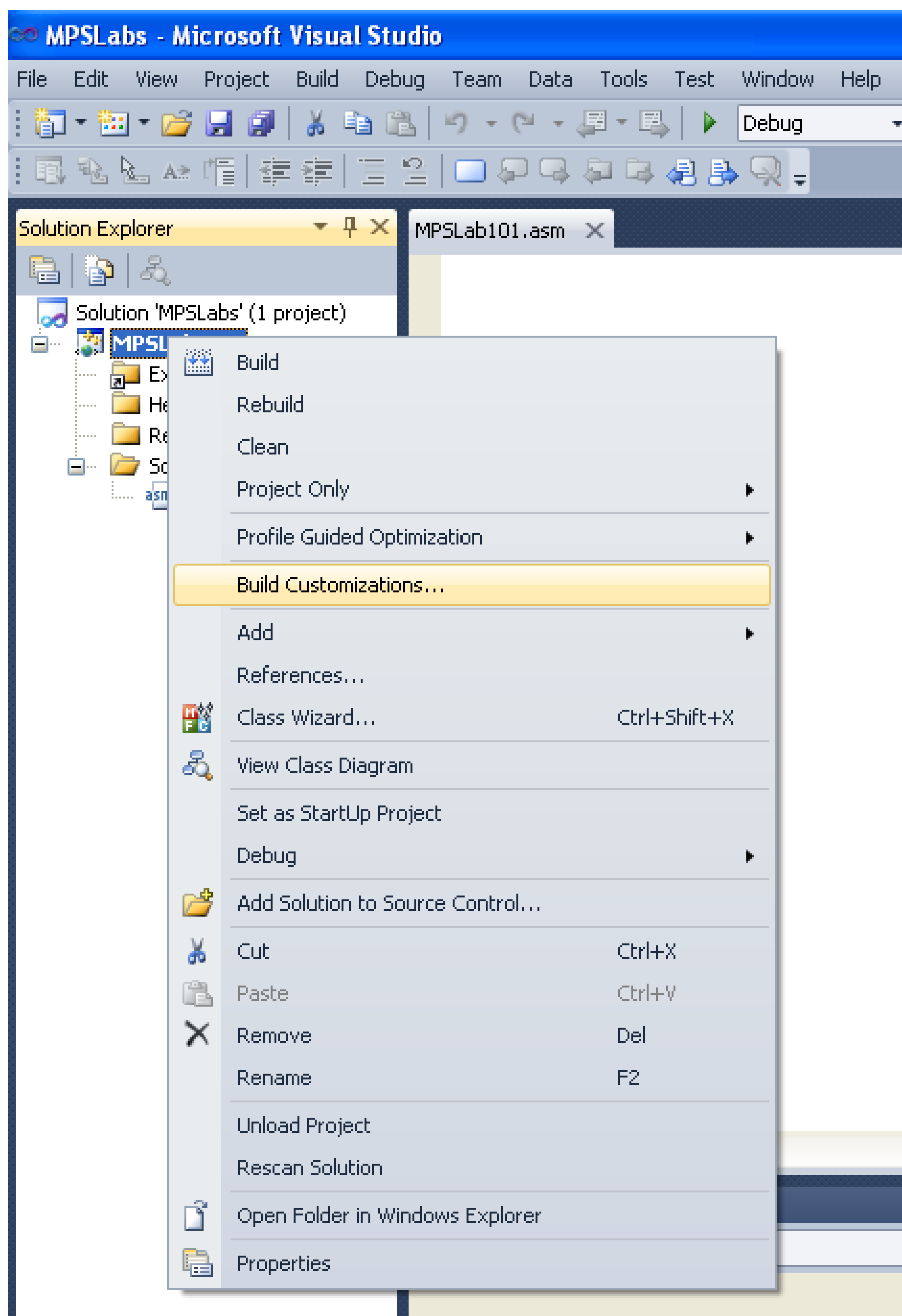


Рисунок 1.3 – Настройка правил для компиляции asm-файлов

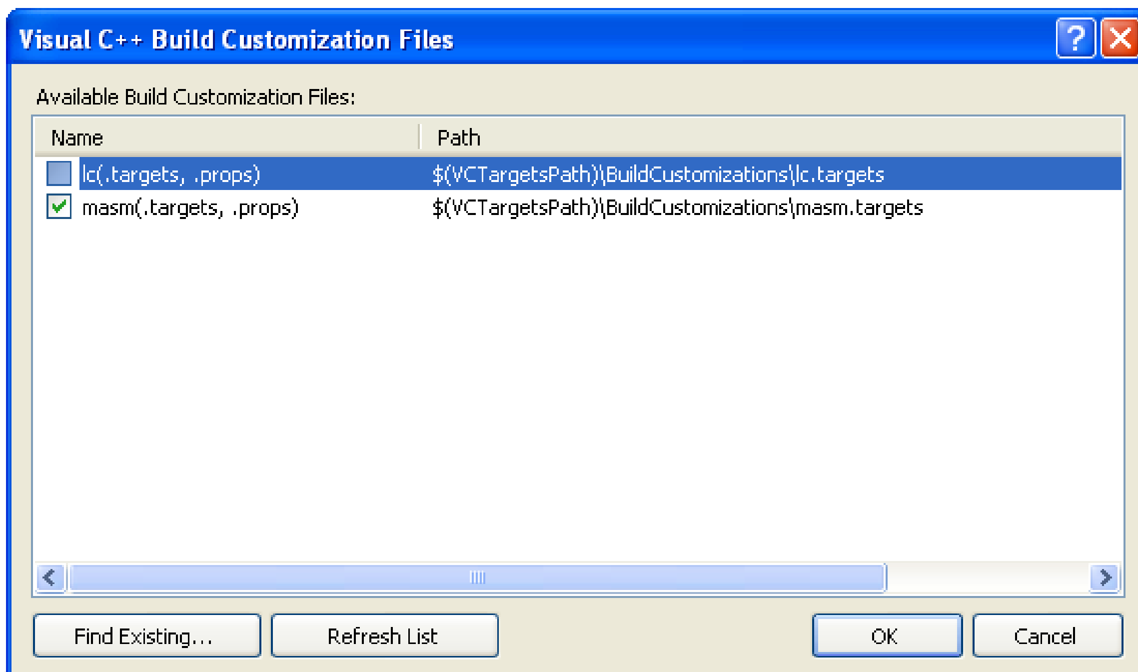


Рисунок 1.4 – Выбор правила для компиляции asm-файлов

2.1.3 Добавление asm-файла в проект

Слева в верхнем углу располагается «Проводник решений» (Solution Explorer), который отображает все файлы, связанные с данным проектом. Файлы разделены на несколько групп. Нас будет интересовать только одна из них: файлы исходного кода (Source Files). Добавим новый файл при помощи пункта меню Новый элемент (New Item) (рисунок 1.5). Выберем тип файла «Текстовый файл (.txt)» (Text File (.txt)) и назовём его, например, MPSLab101.asm (рисунок 1.6).

> *Примечание.* Указание расширения *.asm является важным моментом в настройке проекта. При его отсутствии настройка будут недоступны.

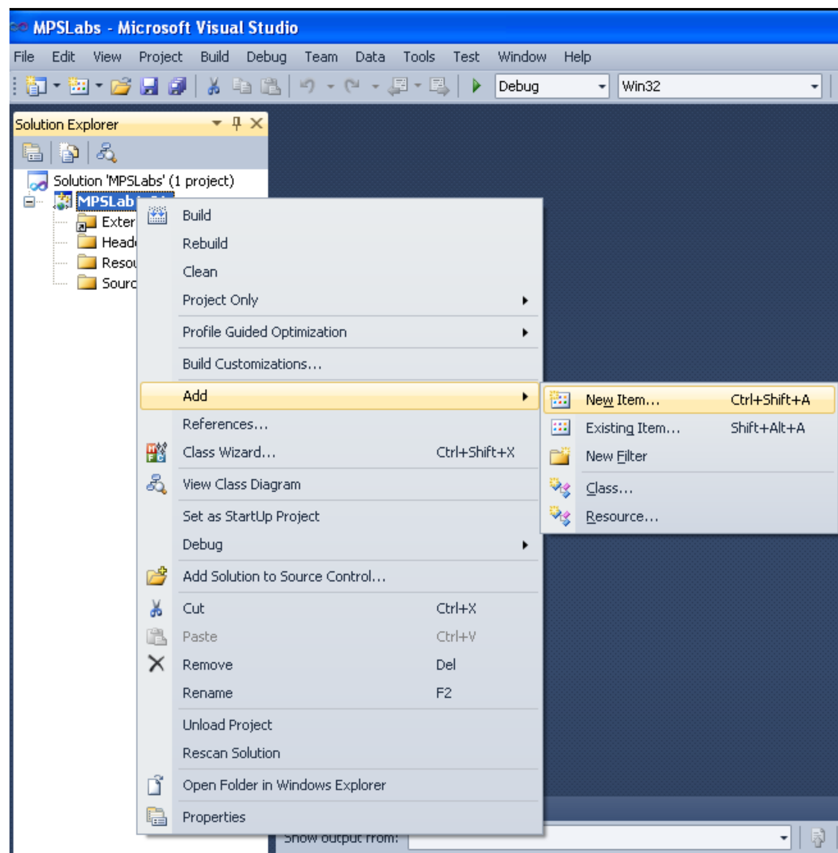


Рисунок 1.5 – Добавление нового файла в проект

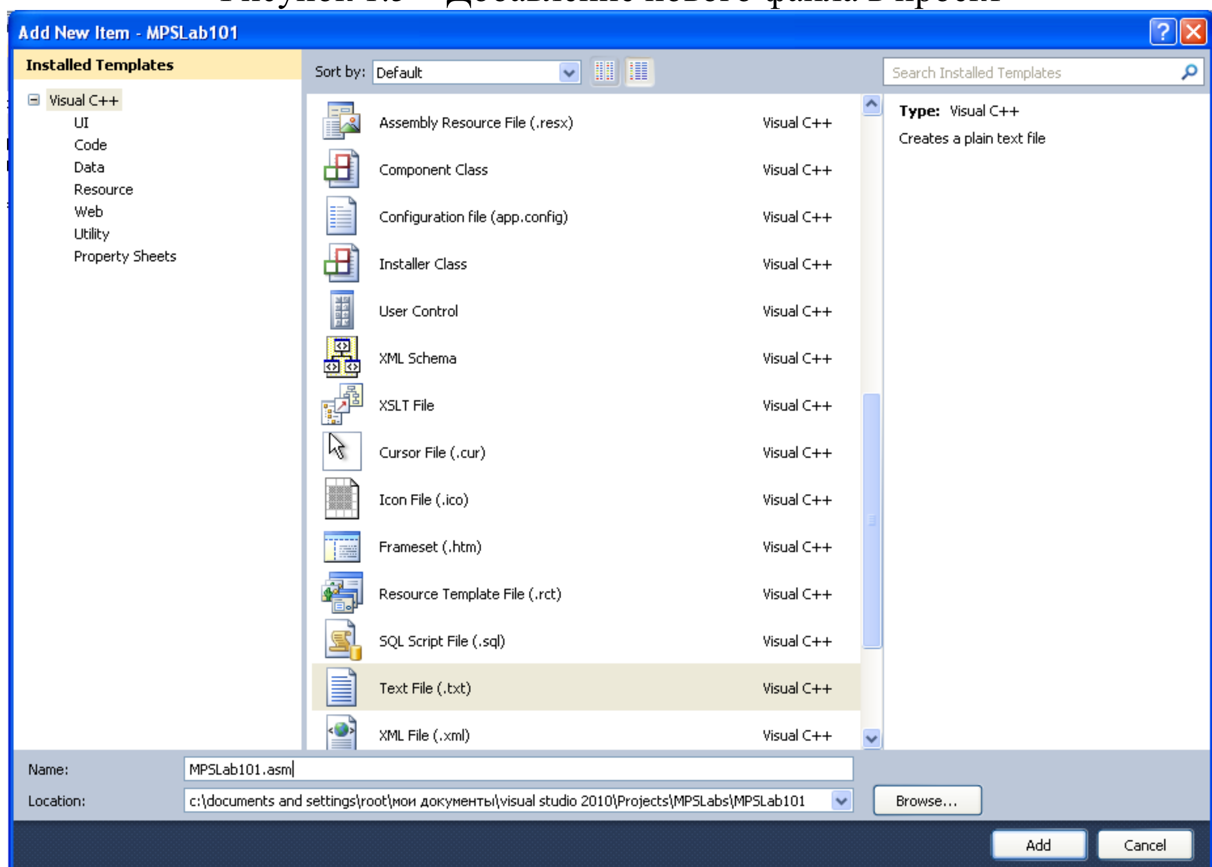


Рисунок 1.6 – Добавление asm-файла

После этого в свойствах проекта должен появиться пункт «Microsoft Macro Assembler» (рисунок 1.7). В противном случае возникла какая-то ошибка.

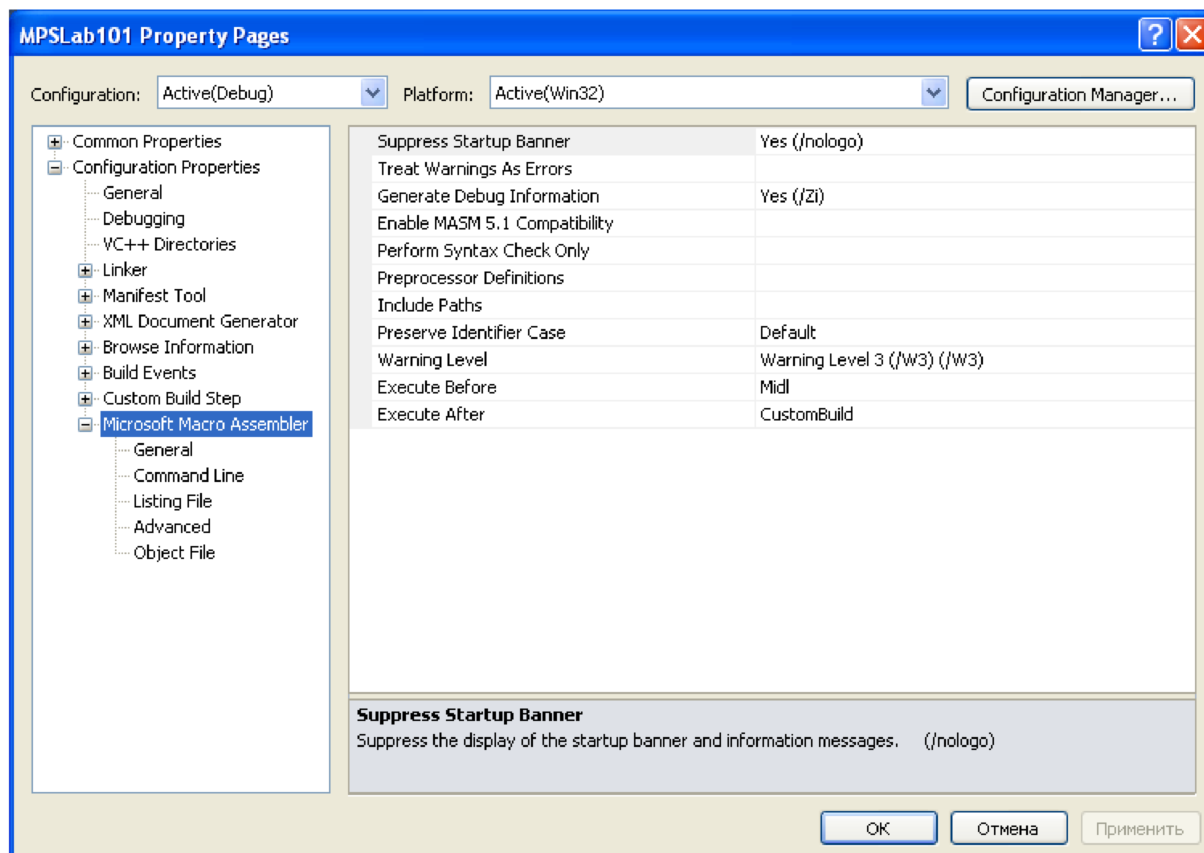


Рисунок 1.7 – Меню настройки masm

2.1.4 Настройка ассемблера (assembler) и компоновщика (linker)

При разработке программ на языке ассемблера компиляция программы состоит из двух этапов: непосредственно компиляции (или ассемблирования) и компоновки. Соответственно в состав среды разработки должны входить оба средства, которые необходимо должным образом сконфигурировать.

Ассемблер (программа) это транслятор с языка ассемблера (компилятор языка ассемблера). Он представляет собой программу, переводящую исходный текст программы с языка, понятного программисту на язык, понятный компьютеру (машинный код). Так как программы обычно составляются из нескольких частей, написанных независимо друг от друга и, зачастую, на разных языках и разными программистами, то компилятор любого языка из любого пакета создает так называемые объектные модули, т.е. файлы специального формата, содержащие помимо машинного кода другую информацию, необходимую для работы компоновщика и отладчика. В DOS и Windows такие файлы получают расширение .obj, а в Unix-системах – .o.

Компоновщик (linker) представляет собой программу, предназначенную для создания исполняемой программы из объектных модулей и библиотечных файлов.

Для работы будет использоваться внешний компилятор masm, входящий в состав Microsoft Visual Studio. Однако необходимые библиотеки и заголовочные файлы необходимо установить отдельно. Их можно бесплатно скачать с сайта проекта http://www.masm32.com.

После установки компилятора и требуемых пакетов необходимо настроить Visual Studio для работы с ним.

Настройка ассемблера заключается в указании путей к включаемым файлам (include files) из состава компилятора masm32. Для этого необходимо указать путь к папке «\masm32\include» в поле «Include Paths» (рисунок 1.8). Это реализуется путем выбора пункта «Изменить» («Edit») в выпадающем меню поля. Пример корректной настройки данного поля показан на рисунке 1.9.

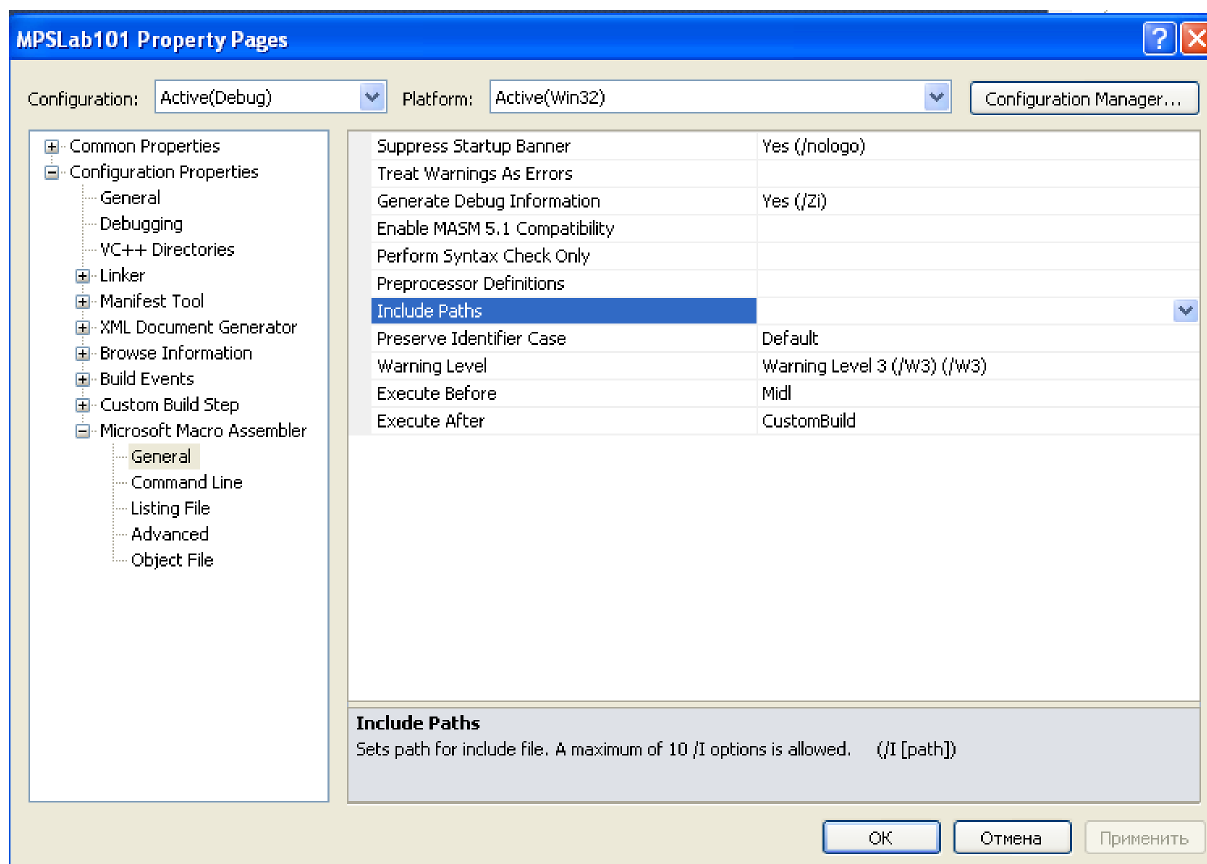


Рисунок 1.8 – Настройка путей для поиска inc-файлов

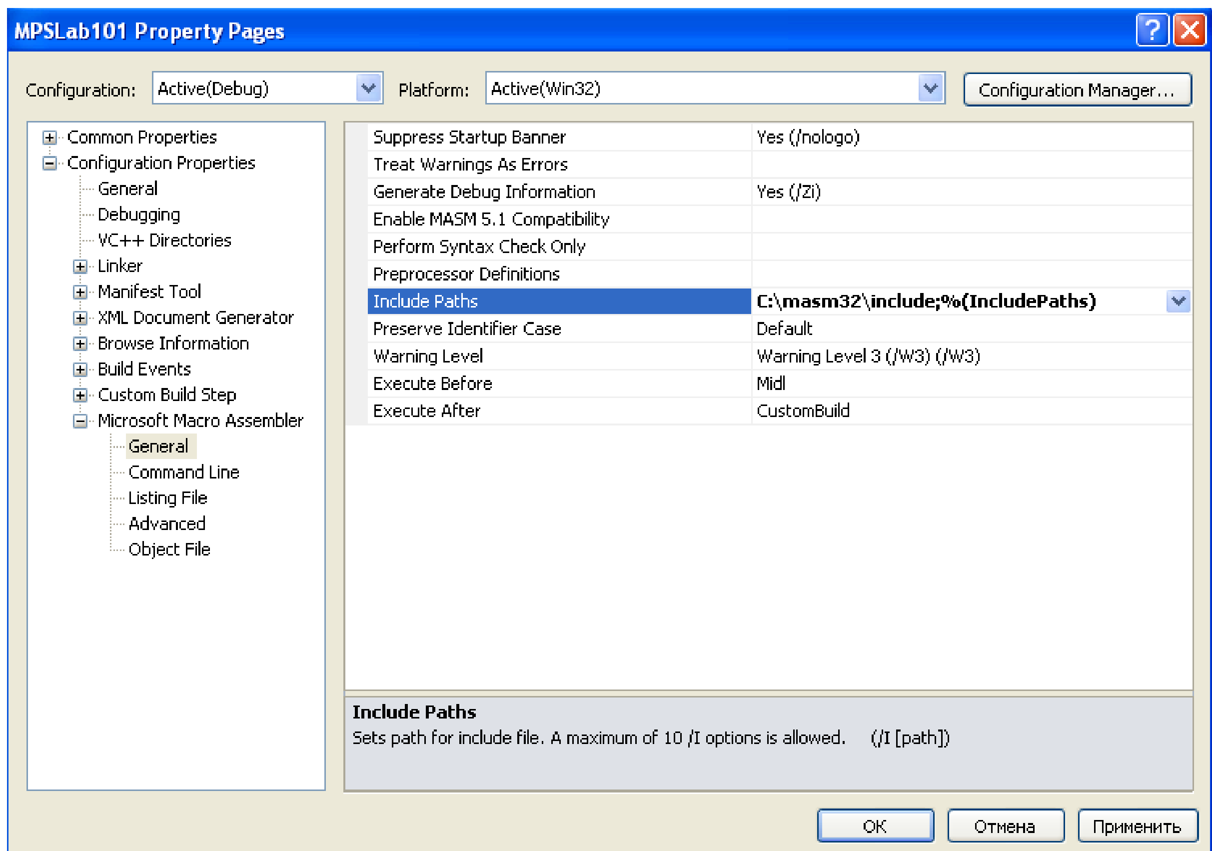


Рисунок 1.9 – Пример настройки путей включения

Теперь необходимо настроить компоновщик.

Во-первых, нужно выбрать систему (SubSystem), для которой будет компоноваться исполняемый файл. Это осуществляется в поле SubSystem, пункта настройки «Linker => System» (рисунок 1.10). Во время выполнения практических занятий будут использоваться в основном вариант «Console (/SUBSYSTEM:CONSOLE)».

Далее необходимо указать точку входа в программу. Это может быть любая валидная метка языка, но обычно используют названия main или start. Точка входа указывается в поле «Entry Point» пункта «Linker => Advanced» (рисунок 1.11).

Далее необходимо указать пути к библиотечным файлам. Для этого нужно указать путь к папке «\masm32\lib» в поле «Additional Library Directories» в разделе «Linker => General», аналогично тому, как это было выполнено для inc-файлов (рисунок 1.12).

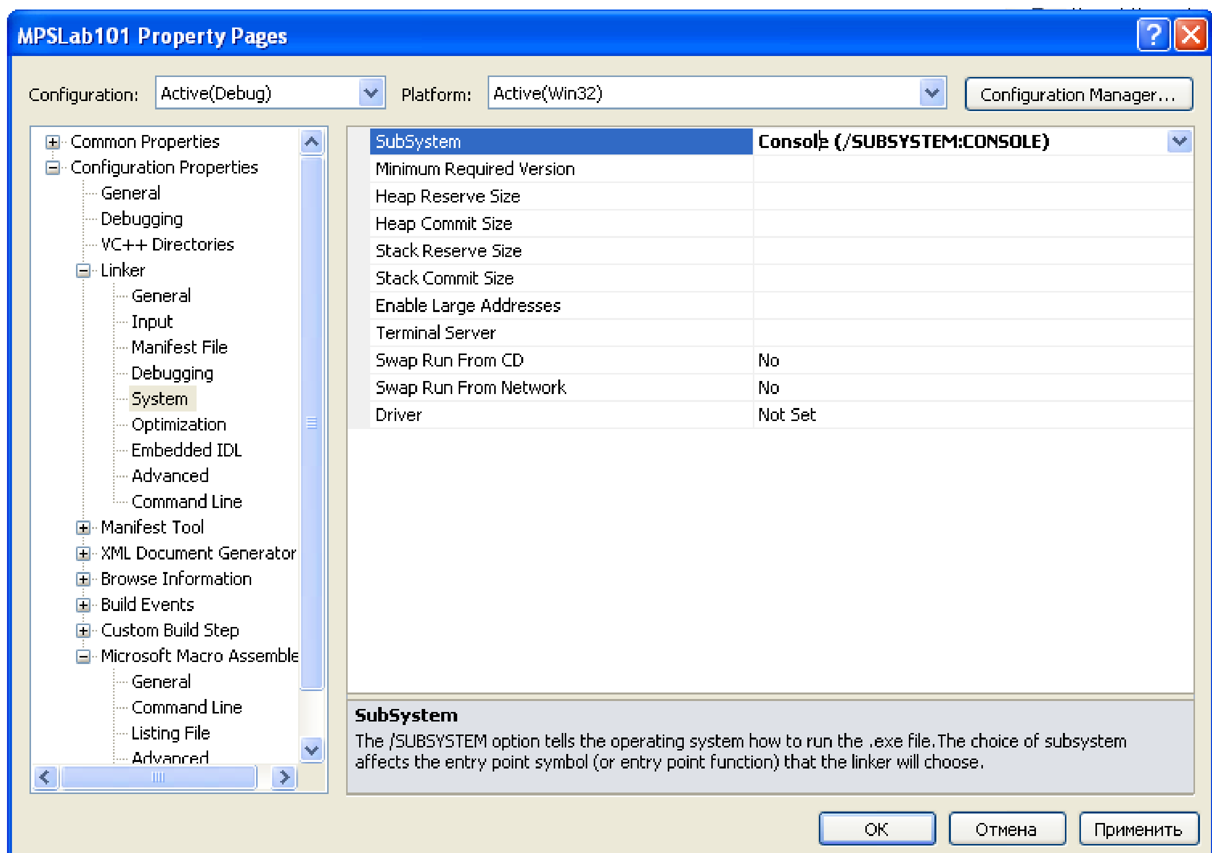


Рисунок 1.10 – Настройка системы исполняемого файла

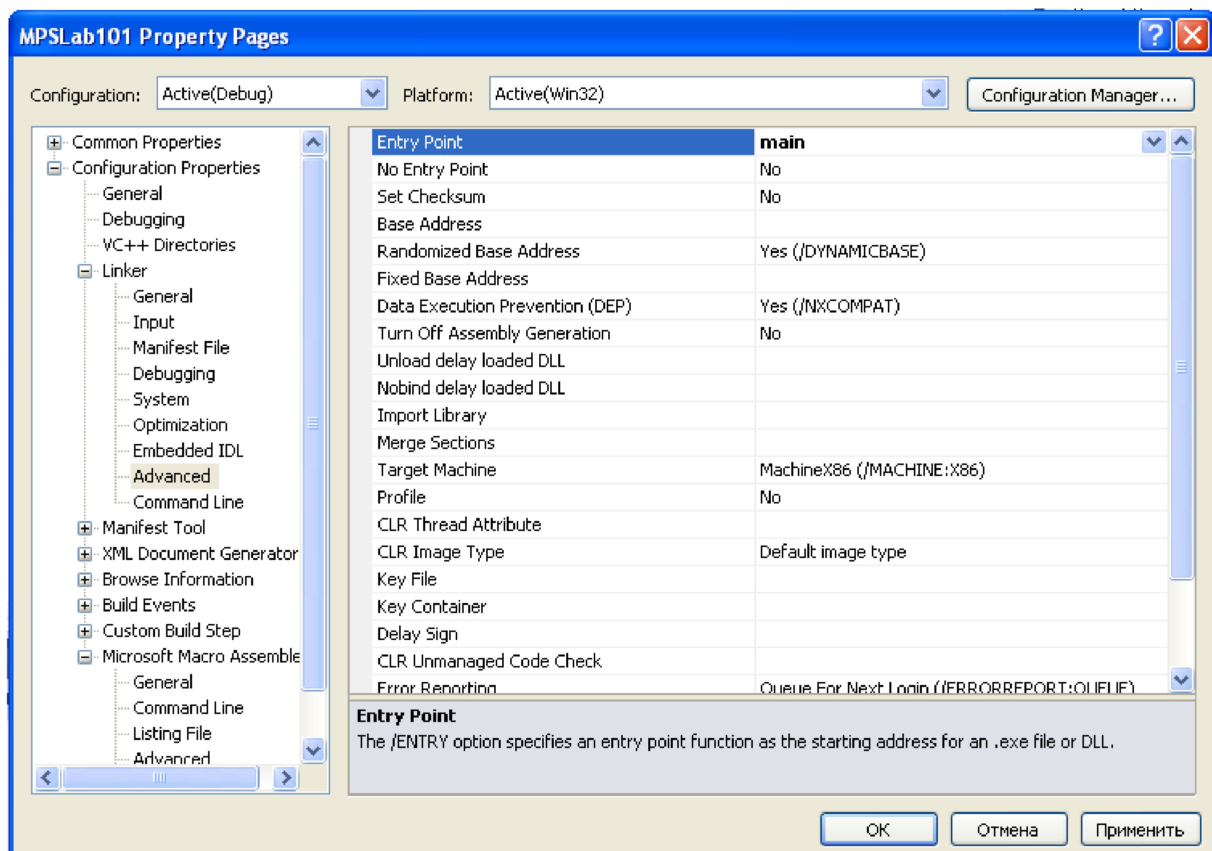


Рисунок 1.11 – Настройка точки входа (Entry Point)

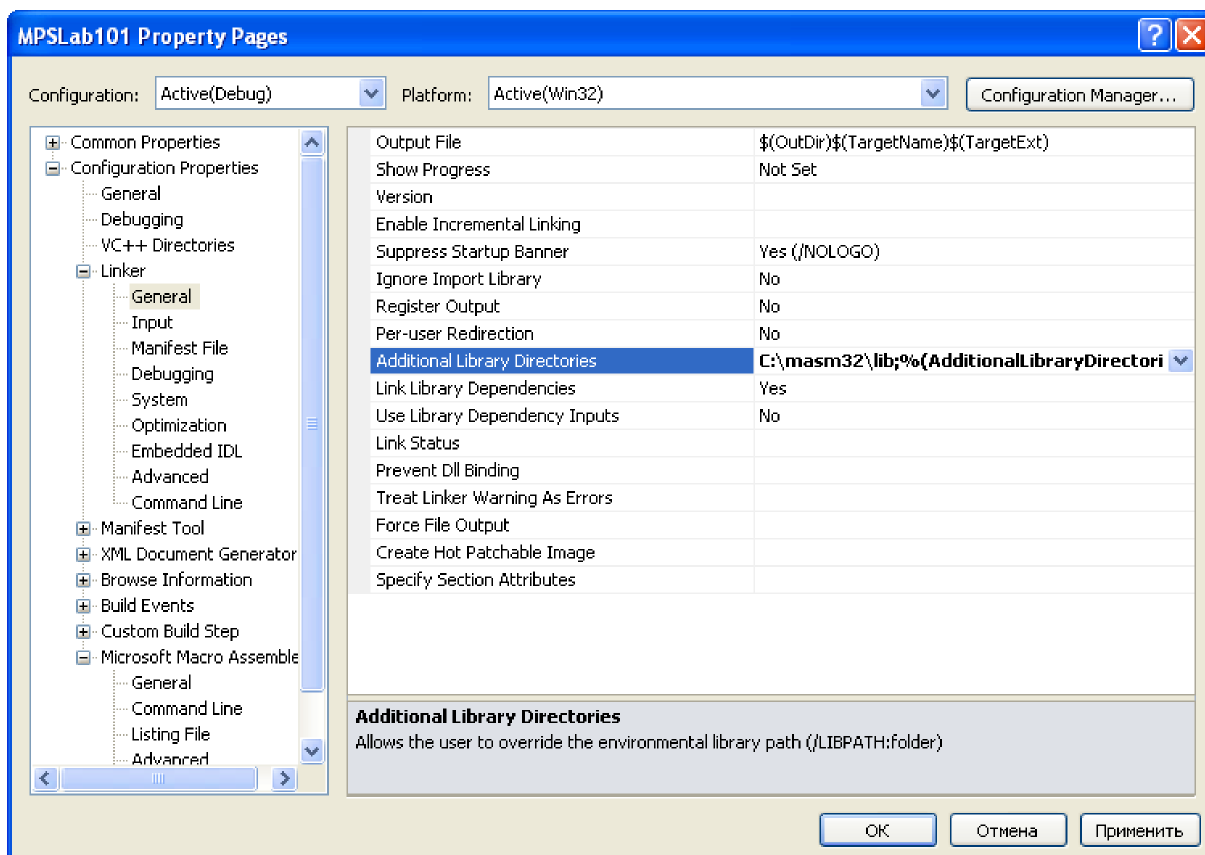


Рисунок 1.12 – Пример настройки путей подключения библиотек

Настройки среды Microsoft Visual Studio 2012 и 2013 практически не отличаются от приведенных выше, однако для ряда программ может потребоваться изменение значения параметра компилятора SAFESEH. Для нормальной работы необходимо в параметре «Image Has Safe Exception Handlers» (пункт «Linker => Advanced») установить значение No (/SAFESEH:NO) (рисунок 1.13).

Рисунок 1.13 – Пример настройки параметра компилятора SAFESEH

2.1.5 Компиляция и запуск программы

Для того чтобы скомпилировать и запустить приложение существует несколько способов. Это и соответствующие пункты меню, и кнопки на панели инструментов, и сочетания горячих клавиш, ускоряющих работу программиста. Рассмотрим доступные варианты подробней (в скобках

указанно сочетание клавиш для среды разработки с настройками для C++, для настроек других языков сочетания могут отличаться).

Build Solution (<F7>) – собрать проект. При этом перекомпилируются все файлы проекта.

Rebuild Solution (<Ctrl + Alt + F7>) – пересобрать проект.

Clean Solution – очистить проект. При этом удаляются все лишние файлы, необходимые на момент разработки и отладки, но не нужные в конечном продукте.

Compile (<Ctrl + F7>) – скомпилировать проект. При этом перекомпилируются только измененные файлы проекта.

Start Debugging (<F5>) – начать отладку. Запускает программу под отладчиком.

Start without Debugging (<Ctrl + F5>) – запустить без отладчика. Просто осуществляется запуск откомпилированной программы.

Step Into (<F11>) – Пошаговое выполнение с заходом в процедуру.

Step Over (<F10>) – Пошаговое выполнение без захода в процедуру.

Toggle Breakpoint (<F9>) – Установить/снять точку останова.

Breakpoints (<Alt + F9>) – показать текущие точки останова.

Теперь попробуем написать первое приложение на языке ассемблера и скомпилировать его.

Откройте в окне редактора файл MPSLab101.asm (если он не был открыт средой разработки автоматически) дважды кликнув на нем в окне проводника решений. Введем текст программы, представленный в примере 1-1, скомпилируем и запустим его.

Пример 1.1. Программы Hello World на языке ассемблера

```
title MPSLab01x01
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc
    include     \masm32\include\masm32.inc

    include      \masm32\macros\macros.asm

    includelib   \masm32\lib\masm32.lib
    includelib   \masm32\lib\kernel32.lib
    includelib   \masm32\lib\user32.lib

; Сегмент данных
.data
; ----------------------------------------------------
; Здесь располагаются данные (переменные) программы
```

```
; -----------------------------------------------------

; Сегмент кода
.code
; Точка входа в программу
main proc far

    ; Макрос print выводит строку на экран
    print "Hello world"

    ; Завершение программы через вызов
    ; системной функции ExitProcess
    invoke ExitProcess, EAX
main endp
; Конец программы
end main
```

2.1.6 Отладка и получение результатов работы программы

Приведенный выше пример использует множество нестандартных расширений и библиотек для организации ввода/вывода и упрощения других рутинных операций. Однако изучение и использование этих расширений не входит в задачу курса.

Сам язык ассемблера не имеет встроенных средств ввода/вывода информации, так как по своей сути представляет собой мнемоническое представление команд микропроцессора. Поэтому основным средством получения информации о результатах работы программы является отладчик. Рассмотрим использование отладчика Visual Studio для получения результатов выполнения программы.

Пример 1.2. Пример программы на языке ассемблера

```
title MPSLab01x02
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib   \masm32\lib\kernel32.lib
    includelib   \masm32\lib\user32.lib

; Сегмент данных
.data
x   dd  87654321h
; Сегмент кода
.code
; Точка входа в программу
main proc far
```

```
        mov EAX, 12345678h
        mov EBX, [x]

    ; Выход из программы
    quit:
        mov EAX, 0
        invoke ExitProcess, EAX
    main endp
    ; Конец программы
    end main
```

Скомпилируем данный пример и запустим его в режиме пошагового выполнения (<F10>).

Для того чтобы увидеть содержимое регистров необходимо отрыть окно Регистры (Registers) через пункт меню Отладка (Debug) – Окна (Windows) – Регистры (Registers) (рисунок 1.14). В него можно добавить отображение дополнительных регистров и флагов (рисунок 1.15). Окно Регистры (Registers) показано на рисунке 1.16.

> *Примечание*. В случае отсутствия пунктов меню Регистры (Registers) необходимо сбросить настройки Visual Studio через пункт меню «Tools => Import and Export Settings …» выбрав вариант «Reset all settings» и далее выбрать настройки среды разработки для языков C# или C++.

Окно содержит пары знамений `имя_регистра = значение`, при этом занесения представлены в 16-ричной системе счисления. Изменяемые значения содержимого регистров и флагов подсвечиваются красным цветом.

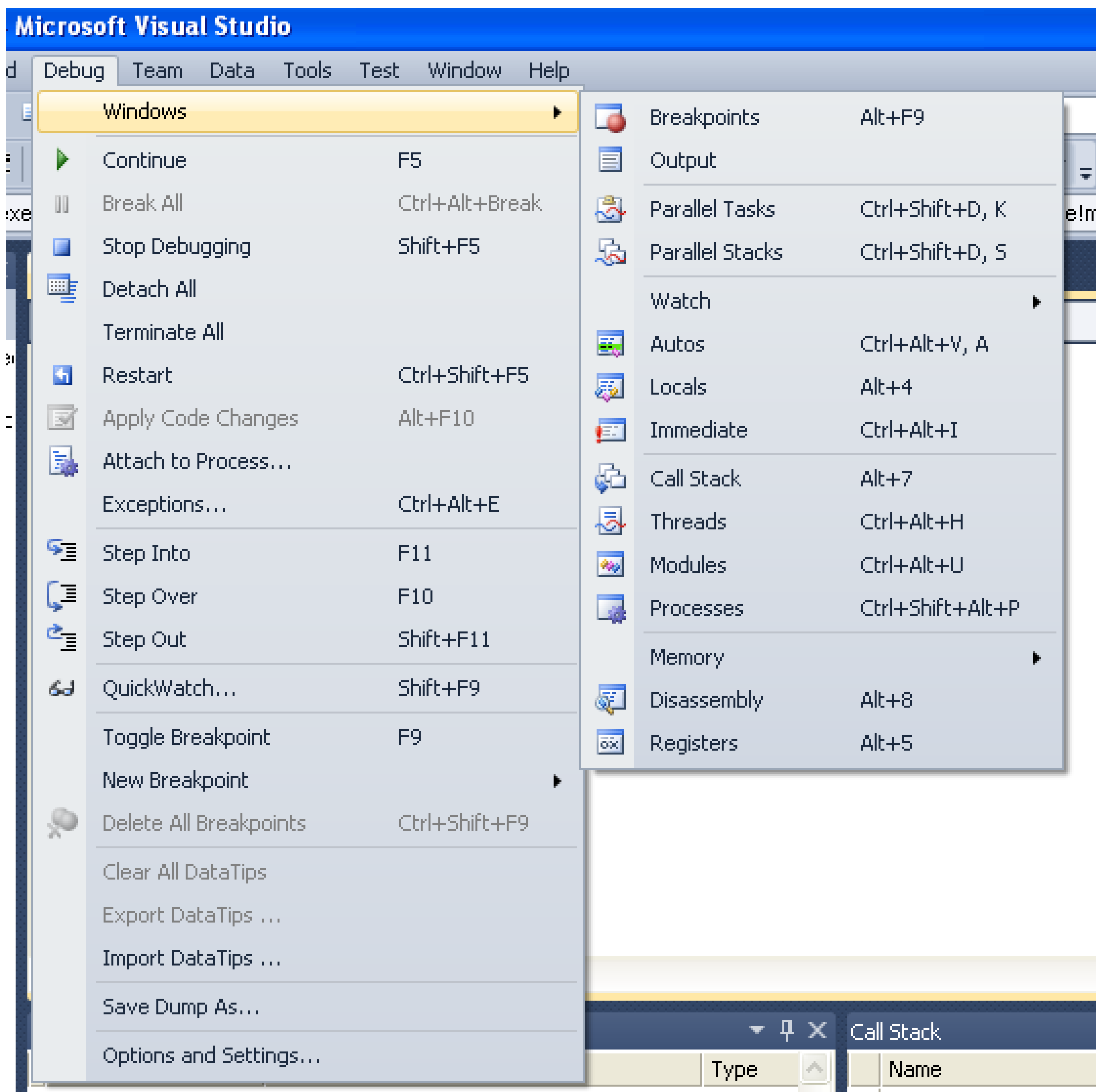


Рисунок 1.14 – Открытие окна Регистры (Registers)

```
Disassembly   MPSLab101.asm
 4
 5  ; Раздел подключения библиотек
 6      include     \masm32\include\windows.inc
 7      include     \masm32\include\kernel32.inc
 8      include     \masm32\include\user32.inc
 9
10      includelib  \masm32\lib\kernel32.lib
11      includelib  \masm32\lib\user32.lib
12
13  ; Сегмент данных
14  .data
15  x   dd  87654321h
16
17  ; Сегмент кода
18  .code
19  ; Точка входа в программу
20  main:
21      mov eax, 12345678h
22      mov ebx, [x]
23
24      ; Выход из программы
25  quit:
26      mov eax, 0
27      invoke ExitProcess, eax
28  ; Конец программы
29  end main
100 %
```

```
Copy              Ctrl+C
Select All        Ctrl+A
✓ CPU
CPU Segments
Floating Point
MMX
SSE
SSE2
3DNow!
Register Stack
Application
Branch
AVX
AVX Float
✓ Flags
✓ Effective Address
```

```
Registers
EAX = 00000000 EBX = 7FFDA000 ECX = 0012FFB0 EDX = 7C90E4
EDI = 00000020 EIP = 00401000 ESP = 0012FFC4 EBP = 0012FF

OV = 0 UP = 0 EI = 1 PL = 0 ZR = 1 AC = 0 PE = 1 CY = 0

Autos  Locals  Registers  Threads  Modules  Watch 1
```

Рисунок 1.15 – Добавление флагов и регистров в окно

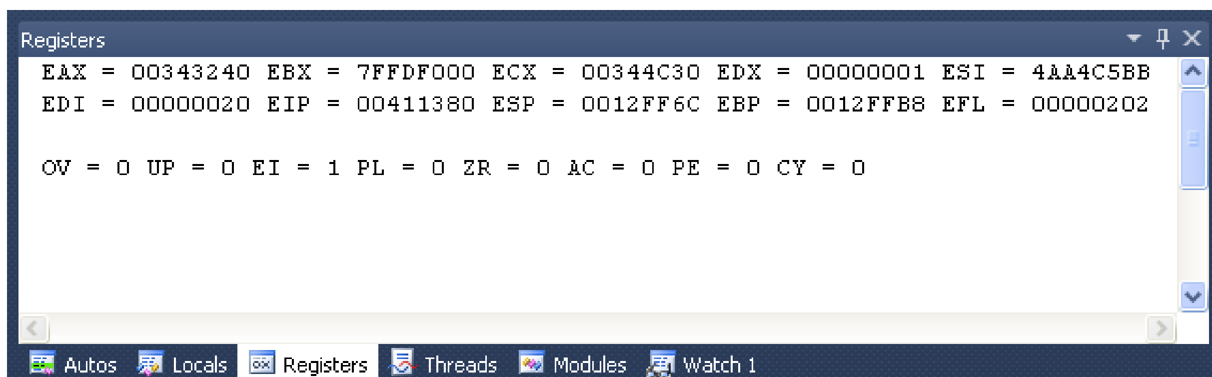


Рисунок 1.16 – Окно Регистры (Registers)

Кроме этого имеется возможность увидеть дизассемблированный код полученной программы. Для этого используется окно Дизассемблирование (Disassembly) (рисунок 1.17)

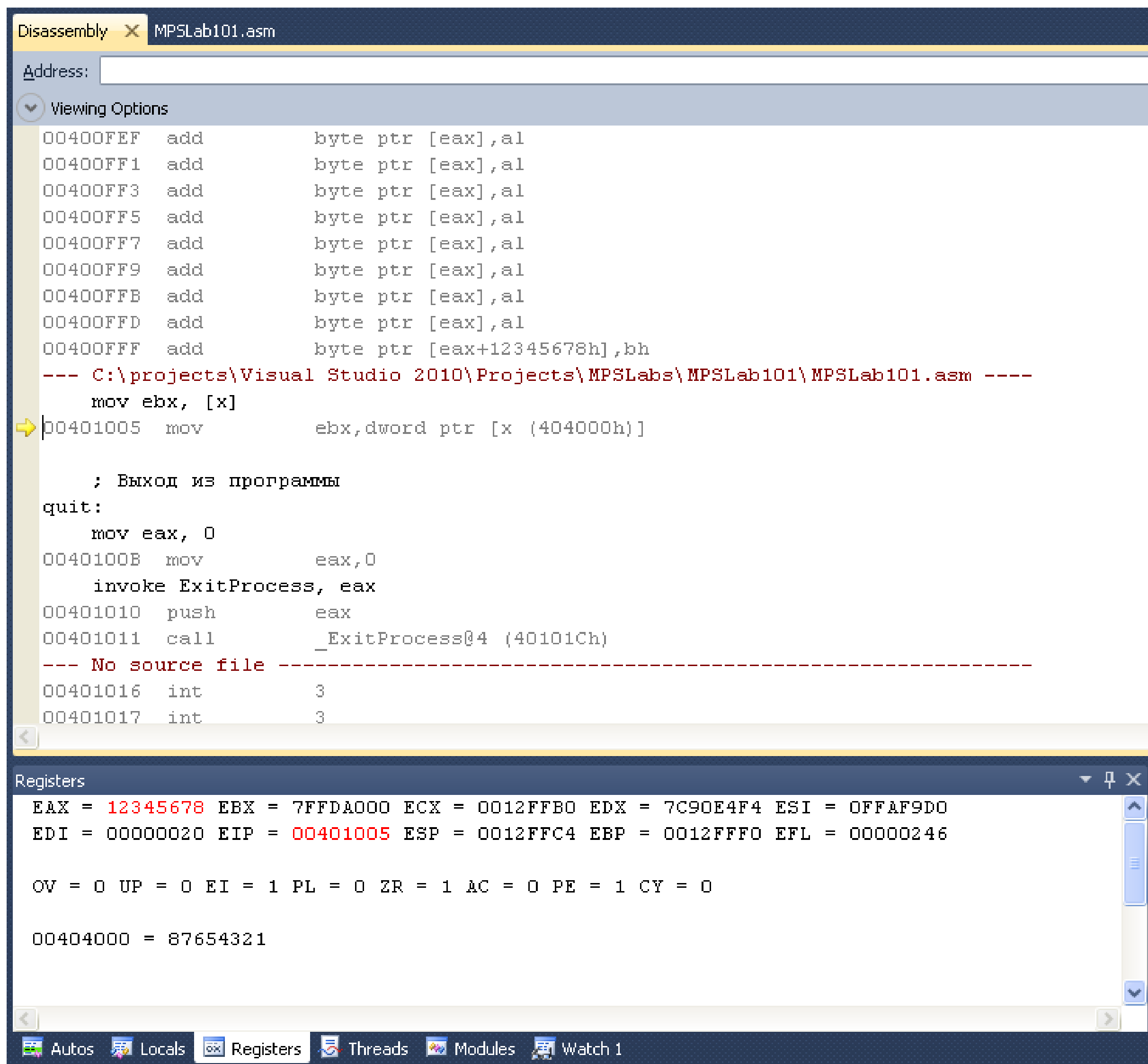


Рисунок 1.17 – Окно Дизассемблирование (Disassembly)

Окна Регистры (Registers) и Дизассемблирование (Disassembly) доступны только в режиме отладки или пошагового выполнения.

2.2 Структура программы и команд на языке ассемблера

2.2.1 Структура программы на языке ассемблера

Немного усложним пример и разберем подробней его содержимое.

Пример 1.3. Структура программы на языке ассемблера

```
title MPSLab01x03
.686
.model flat, stdcall
option casemap :none    ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
```

```
    include     \masm32\include\user32.inc

    includelib   \masm32\lib\kernel32.lib
    includelib   \masm32\lib\user32.lib

; Сегмент данных
.data
x   dd  87654321h

; Сегмент кода
.code
; Точка входа в программу
main proc far

    mov EAX, 12345678h  ; Поместим число 0x12345678 в регистр EAX

    add EAX, 0CC796837h ; Прибавим к содержимому EAX число 0xCC796837
                        ; Результат помещается в EAX

    jmp quit            ; Переход на метку quit

    ; Операторы находящиеся ниже будут пропущены
    mov EBX, [x]        ; Поместим содержимое переменной x в регистр EBX
    sub EAX, EBX        ; Отнимем от содержимого EAX содержимое EBX

    ; Выход из программы
quit:
    mov EAX, 0
    invoke ExitProcess, EAX
main endp
; Конец программы
end main
```

Первые строчки определяют набор инструкций процессора, используемую модель памяти, чувствительность к регистру символов.

Далее идут директивы, включающие включаемые (inc) файлы, используемые для подключения библиотек и других файлов проекта. Данный набор является базовым для программы.

После этого следует объявление сегмента данных при помощи директивы `.data`. В нем располагаются переменные (глобальные), используемые в программе. Данный сегмент продолжается до объявления следующего сегмента.

Следом за ним идет сегмент кода, определяемы при помощи директивы `.code`. В нем располагается код программы на языке ассемблера. Точкой входа программы является `main`. Она первой получает управление при запуске программы.

Функция заканчивается вызовом функции ExitProcess, завершающей программу и возвращающей управление обратно операционной системе.

Конец программы определяется директивой `end`, за которой следует имя точки входа программы.

В общем виде шаблон программы представлен в примере 1.4

Пример 1.4. Шаблон программы с ассемблерными вставками

```
title MPSLab01x04
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib   \masm32\lib\kernel32.lib
    includelib   \masm32\lib\user32.lib

; Сегмент данных
.data
; ---------------------------------------------------
; Здесь располагаются данные (переменные) программы
; ---------------------------------------------------

; Сегмент кода
.code
; Точка входа в программу
main proc far
; ---------------------------------------------------
; Здесь располагается код программы
; ---------------------------------------------------

    ; Выход из программы
quit:
    mov EAX, 0
    invoke ExitProcess, EAX
main endp
; Конец программы
end main
```

2.2.2 Структура команд на языке ассемблера

Рассмотрим подробней структуру команд на языке ассемблера. Ее можно представить условно состоящей из четырех частей:

```
[метка:] инструкция параметры  [; комментарии]
```

> *Примечание*. Здесь заключение символов в квадратных скобках означает необязательность использования, а не элемент синтаксиса.

При этом в каждой строке может располагаться только одна команда, представляющая собой одну инструкцию микропроцессора.

В начале команды идет метка, представляющая собой символьное имя для адреса данной команды. Метка является необязательным элементом команды, а также может находиться одна в строке, указывая при этом на первую следующую за ней команду (например, как предыдущем примере метка `quit`).

Далее следует мнемонический код (символьное представление) инструкции микропроцессора, за которым, через пробел, идут параметры, разделяемые запятыми.

Все символы, следующие после точки с запятой до конца строки, являются комментарием и игнорируются компилятором.

Переменные не могут быть объявлены непосредственно внутри ассемблерного кода, так как значения переменных будут интерпретированы как двоичные коды команд. Для объявления переменных используются отдельный сегмент данных.

Синтаксис объявления переменных похож на синтаксис команды:

```
имя инструкция_выделения_памяти значение  [; комментарии]
```

Часть «значение» задает начальное значение переменной. Если значение заранее не известно, то указывается знак вопроса.

Инструкции выделения памяти определяют размер выделяемой под хранение данных. При описании данных и их типов (размеров) используются следующие директивы:

db – байт (8 бит);

dw – слово (16 бит);

dd – двойное слово (32 бит);

dq – учетверенное слово (64 бит);

dt – 10 байт (80 бит).

Тип db также используется для описания символов и строк в кодировке ASCII.

Типы dd, dq и dt используются для описания чисел с плавающей точкой математического сопроцессора или FPU.

Для резервирования некоторого количества памяти используется директива dup.

```
имя инструкция_выделения_памяти количество dup (начальное_значение)
```

При объявлении целочисленных констант в программе могут использоваться следующие суффиксы, указывающие на систему счисления:

b – двоичное целое число;

o или q – восьмеричное целое число;

d – десятичное целое число (по умолчанию);

h – шестнадцатеричное целое число.

Шестнадцатеричная система кроме своего прямого назначения также используется для записи целых чисел в форматах BCD и ASCII.

Переменные на языке ассемблера не являются типизированным в смысле языков высокого уровня. Тип, по сути, является лишь указанием на количество байт, необходимых для хранения данных. При работе с переменными контролируется только соответствие размеров переменных (которое можно обойти путем использования соответствующих директив). На самом деле имя переменной является указателем на первую выделенную ячейку памяти.

Пример 1.5. Примеры описания данных

```
x       db 11110000b    ; Число 240 в двоичной системе счисления
y       dw 0FFFFh       ; Число 65535 в шестнадцатеричной системе
z       dd 100          ; Число 100 в десятичной системе счисления
f       dd 3.75         ; Число с плавающей точкой одинарной точности
d       dd ?            ; Переменная не инициализированна значением
a       db 30h          ; Символ "ноль" в ASCII формате
b       dw 0105h        ; Число 15 в неупакованном ASCII-формате
msg     db 'Hello!'     ; Строка байт
array   dd 0, 2, 4, 6   ; Массив чисел
buffer  db 256 dup (0)  ; Резервирование 256 байт
```

При обращении к данным в ячейке памяти, в окне Регистры (Registers) появляются значения эффективного адреса переменой и ее значение (рисунок 1.18). (Слева указан эффективный адрес переменной x (ячейки памяти с адресом [00404000]) и значение, находящееся в ней (число 2271560481 (0x87654321) в 16-ричной системе счисления)).

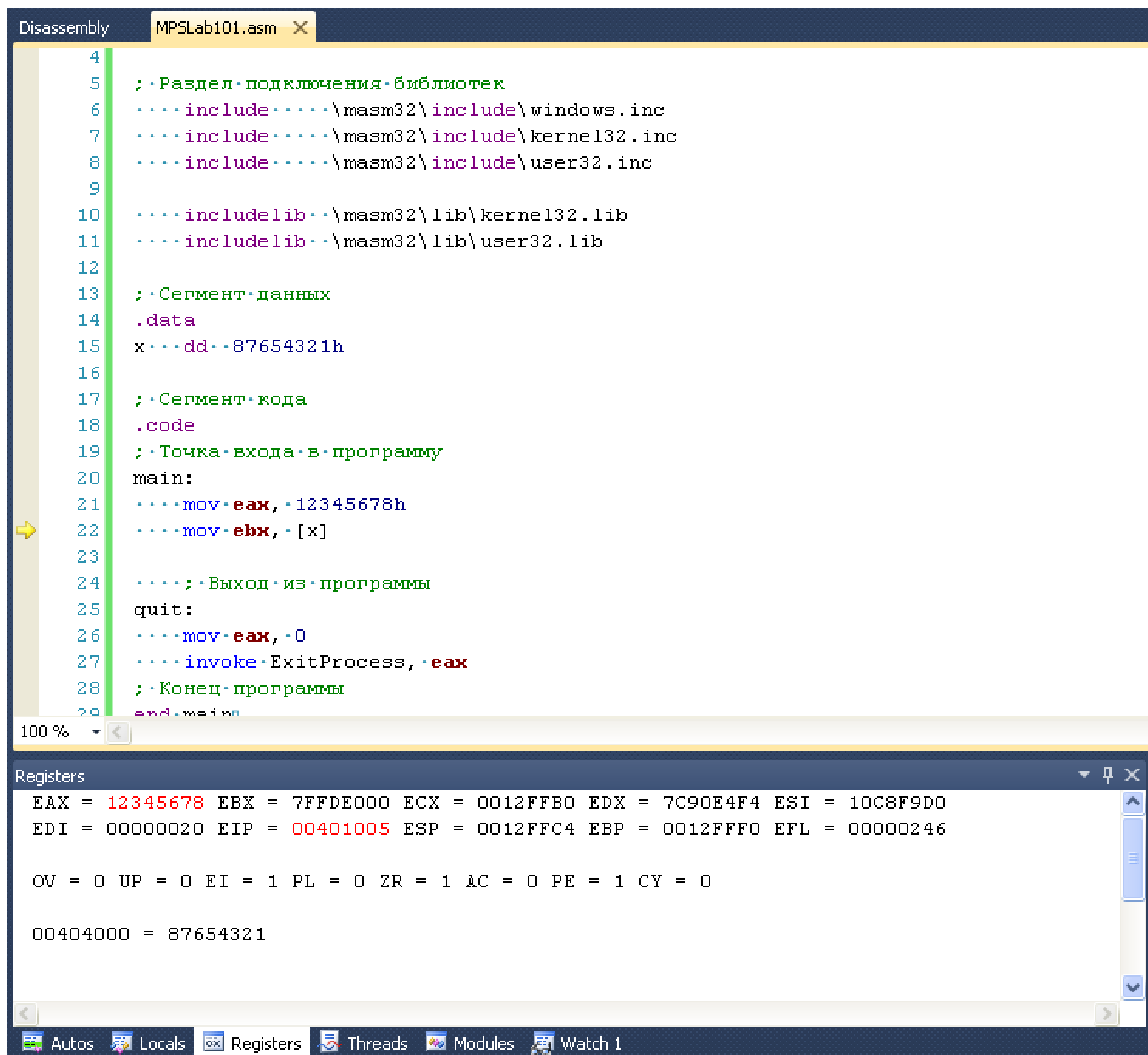


Рисунок 1.18 – Окно Регистры (Registers) с отображением переменных

Пример 1.6. Объявление и использование переменных

```
title MPSLab01x06
.686
.model flat, stdcall
option casemap :none    ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib   \masm32\lib\kernel32.lib
    includelib   \masm32\lib\user32.lib

; Сегмент данных
.data
x   dd   87654321h
y   dd   ?
```

```
; Сегмент кода
.code
; Точка входа в программу
main proc far
    mov EAX, 12345678h  ; Помещаем в регистр EAX число 0x12345678
    mov EBX, [x]        ; Помещаем в регистр EBX значение переменной x
    add EAX, EBX        ;Складываем EAX и EBX и помещаем результат в EAX
    mov [y], EAX        ; Сохраняем результат в переменной y

    ; Выход из программы
quit:
    mov EAX, 0
    invoke ExitProcess, EAX
main endp
; Конец программы
end main
```

2.3 Отладка

Иногда, когда программа делает что-то непредвиденное, причина достаточно очевидна, и можно быстро исправить код программы. Но другие ошибки более трудноуловимы и вызываются взаимодействие различных частей программы. В этих случаях лучше всего остановить программу в заданной точке, пройти ее шаг за шагом и просмотреть состояние переменных и выражений. Такое управляемое выполнение – ключевой элемент отладки.

2.3.1 Установка точки прерывания

Точка прерывания позволяет остановить выполнение программы перед любой выполняемой инструкцией (оператором) с тем, чтобы продолжать выполнение программы либо в пошаговом режиме, либо в непрерывном режиме до следующей точки прерывания.

Чтобы задать точку прерывания перед некоторым оператором, необходимо установить перед ним текстовый курсор и нажать клавишу F9 или кликнуть мышью на левом боковом поле окна редактирования напротив оператора. Точка прерывания обозначается в виде красного кружка на левом поле окна редактирования, а также красным фоном выделяется оператор, при выполнении которого сработает точка остановки (см. рисунок 1.19). Повторное действие (щелчок на указанной кнопке или нажатие F9) снимает точку прерывания. В программе может быть несколько точек прерывания.

Для просмотра всех точек остановки необходимо выбрать пункт Debug | Windows | Breakpoints или нажать комбинацию клавиш Ctrl+Alt+B.

```
 8      include     \masm32\include\user32.inc
 9
10      includelib  \masm32\lib\kernel32.lib
11      includelib  \masm32\lib\user32.lib
12
13  ; Сегмент данных
14  .data
15  x   dd  87654321h
16  y   dd  ?
17
18  ; Сегмент кода
19  .code
20  ; Точка входа в программу
21  main:
22      mov eax, 12345678h
23      mov ebx, [x]
24      add eax, ebx
25      mov [y], eax
26
27      ; Выход из программы
28  quit:
29      mov eax, 0
30      invoke ExitProcess, eax
31  ; Конец программы
32  end main
```

Рисунок 1.19 – Окно редактора с установленной точкой прерывания

2.3.2 Выполнение программы до точки прерывания

Программа запускается в отладочном режиме с помощью команды Debug | Start Debugging (или нажатием клавиши F5).

В результате код программы выполняется до строки, на которой установлена точка прерывания. Затем программа останавливается и отображает в окне Editor ту часть кода, где находится точка прерывания, причем желтая стрелка на левом поле указывает на строку (см. рисунок 1.18), которая будет выполняться на следующем шаге отладки.

2.3.3 Прекращение отладки

В ходе сеанса отладки иногда желательно начать все сначала. Выбор команды Debug | Stop Debugging или нажатие клавиши Shift+F5 приведет к полному сбросу, так что выполнение по шагам, или трассировка прекратится.

2.3.4 Пошаговое выполнение программы и трассировка

Команды выполнения по шагам Step Over (клавиша F10) и трассировки Trace Into (клавиша F11) меню Debug дают возможность построчного выполнения программы. Единственное отличие выполнения по шагам и трассировки состоит в том, как они работают с вызовами функций.

Выполнение по шагам вызова функции интерпретирует вызов как простой оператор и после завершения подпрограммы возвращает

управление на следующую строку. Трассировка подпрограммы загружает код этой подпрограммы и продолжает ее построчное выполнение.

Нажимая клавишу F10, можно выполнять один оператор программы за другим. Предположим, что при пошаговом выполнении программы вы дошли до строки, в которой вызывается некоторая функция `func1`. Если вы хотите пройти через код вызываемой функции, то надо нажать клавишу F11, а если внутренняя работа функции вас не интересует, а интересен только результат ее выполнения, то надо нажать клавишу F10.

Допустим, что вы вошли в код функции `func1`, нажав клавишу F11, но через несколько строк решили выйти из него, т.е. продолжить отладку после возврата из функции. В этом случае надо нажать клавиши Shift+F11 или выбрать пункт Debug | Step Out.

2.3.5 Выполнение программы до курсора

Иногда, конечно, нежелательно выполнять по шагам всю программу только для того, чтобы добраться до того места, где возникает проблема. Отладчик позволяет выполнить сразу большой фрагмент программы до той точки, где необходимо начать выполнение по шагам.

Существует возможность пропустить пошаговое выполнение некоторого куска программы: установите текстовый курсор в нужное место программы и нажмите клавиши Ctrl+F10. Программа будет выполнена до курсора и остановится в данной точке, ожидая дальнейших действий пользователя.

Причем, это можно сделать как в начале сеанса отладки, так и когда уже часть программы выполнена по шагам.

Чтобы продолжить дальнейшее выполнение программы без пошагового режима нажмите F5.

2.3.6 Отслеживание значений переменных во время выполнения программы

Выполнение программы по шагам или ее трассировка могут помочь найти ошибки в алгоритме программы, но обычно желательно также знать, что происходит на каждом шаге со значениями отдельных переменных.

Для наблюдения за выводом программы встроенный отладчик имеет средства для просмотра значений переменных, выражений и структур данных.

Чтобы узнать значение переменной задержите над ней указатель мыши. Рядом с именем переменной на экране появляется подсказка со значением этой переменной.

Часто программисту необходимо отслеживать значение конкретной переменной или выражения при выполнении программы по шагам.

Помимо экранной подсказки, переменные со своим значением могут отображаться в окне «Watch 1», расположенном в левом нижнем углу

экрана (см. рисунок 1.20). В этом окне можно задать имя переменной, значения которой вы хотите отслеживать.

Также можно добавить переменную в список просмотра «Watch 1» нажав правую кнопку мыши над именем переменой и выбрав пункт «Add Watch». Кроме этого в окне просмотра «Watch 1» можно добавлять или удалять отслеживаемые элементы. В этом диалоговом окне можно проверять переменные и выражения и изменять значения любых переменных, включая строки, указатели, элементы массива и поля записей, что позволяет проверить реакцию программы на различные условия.

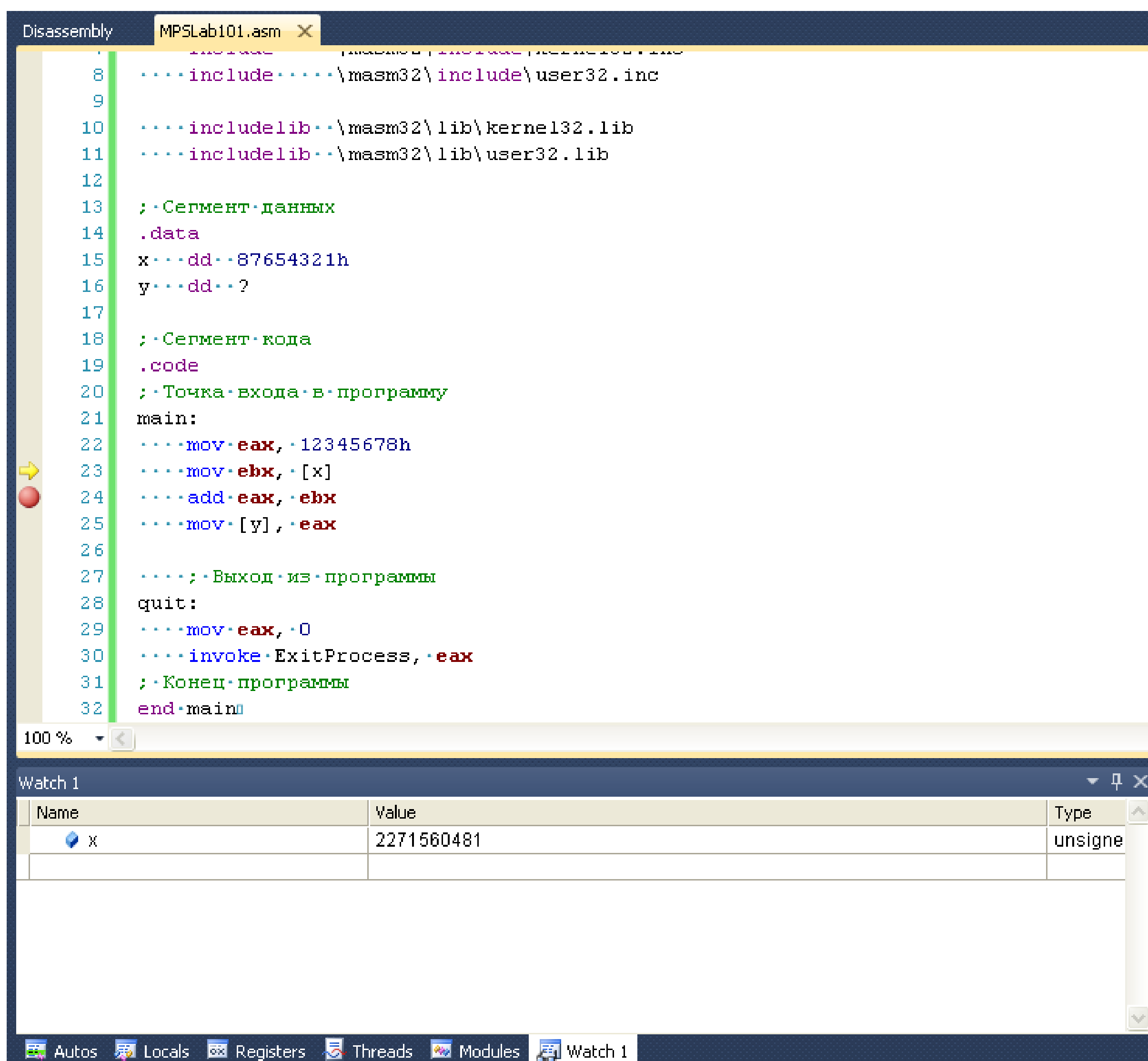


Рисунок 1.20 – Окно редактора и список отслеживаемых переменных

Оба средства вычислений и просмотра работают на уровне выражений, поэтому важно определить, что считается выражением. Выражение состоит из констант, переменных и структур данных, скомбинированных с помощью операций и большинства встроенных функций. Почти все, что можно использовать в правой части оператора

присваивания, может также использоваться в качестве отладочного выражения.

Во время отладки можно изменить значение переменной. Для этого нужно ввести новое значение переменной в столбце Value, после чего оно отобразится красным цветом.

3 Контрольные вопросы

1) Как выполняется настройка среды Visual Studio и проекта для работы с языком ассемблера?
2) Как выполняется компиляция проекта?
3) Опишите структуру приложения на языке ассемблера.
4) Как просмотреть содержимое регистров микропроцессора?
5) Как просмотреть содержимое флагов микропроцессора?
6) Опишите структуру команды на языке ассемблера.
7) Опишите структуру определения переменной на языке ассемблера.
8) Для чего предназначена отладка?
9) Какие средства отладки предоставляет среда разработки MS Visual Studio?
10) Что такое точка остановки (breakpoint)?
11) Какие возможности существуют для слежения за значениями переменных во время отладки?
12) Какие горячие клавиши для работы с отдатчиком вы знаете?

4 Задание

1) Изучить возможности интегрированной среды разработки Microsoft Visual Studio.
2) Настроить среду разработки Microsoft Visual Studio для работы с языком ассемблера.
3) Создать новый проект приложения для работы на языке ассемблера.
4) Настроить конфигурацию проекта для работы с языком ассемблера.
5) Набрать и отладить программу, приведенную в примере 1.1.
6) Пошагово выполнить полученную программу, отображая изменения регистров микропроцессора.
7) Набрать и отладить программу, приведенную в примере 1.2.
8) Пошагово выполнить полученную программу, отображая изменения регистров микропроцессора.
9) Набрать и отладить программу, приведенную в примере 1.3.
10) Пошагово выполнить полученную программу, отображая изменения регистров микропроцессора.
11) Набрать и отладить программу, приведенную в примере 1.6.
12) Пошагово выполнить полученную программу, отображая изменения регистров микропроцессора.
13) Оформить отчёт.

Практическое занятие № 2. Регистры. Организация памяти и режимы адресации. Команды пересылки данных

1 Цель работы

Цель работы – изучить команды пересылки данных, работу с пользовательскими регистрами и режимы адресации.

2 Краткая теория

Программная модель компьютера – часть компьютера оставлена видимой и доступной для программирования. Ее частью является программная модель микропроцессора.

Различают несколько поколений процессоров программно совместимых с микропроцессорами Intel, носящие общее имя x86.

Архитектура IA-16. Процессоры, программно совместимые с Intel 8086-80286 имеют 14 регистров, используемых для управления выполняющейся программой, для адресации памяти и для обеспечения арифметических вычислений. Каждый регистр имеет длину 16 бит (одно слово) и адресуется по имени.

Архитектура IA-32. Процессоры данной архитектуры начали свою историю с МП Intel 80386 и разрабатывались вплоть до появления первых 64-битных компьютеров с архитектурой Intel 64/AMD64 (не путать с IA-64, используемой в МП семейства Itanium). Процессоры семейства x86-32 расширяют большинство основных регистров до 32-х разрядов, оставаясь при этом программно совместимыми с прежними 16-ти разрядными процессорами. Содержат 32 регистра, которые можно разделить на две большие группы:

- 16 пользовательских регистров;
- 16 (15 + 1 резерв) системных регистров.

2.1 Регистры и память

При разработке архитектуры компьютера относительно регистров принято использовать 2 подхода: регистры могут быть универсальными (могут использоваться для нескольких команд) или использоваться только для определенных команд. Большинство регистров МП x86 имеют определенное функциональное назначение.

Биты регистра принято нумеровать слева направо (младшие биты справа, а старшие слева) (рисунок 2.1). Регистры это сверхбыстрые ячейки памяти входящие в состав микропроцессора, используемые для выполнения операций над данными и краткосрочного хранения информации. Минимально адресуемой единицей информации внутри микропроцессора является бит.

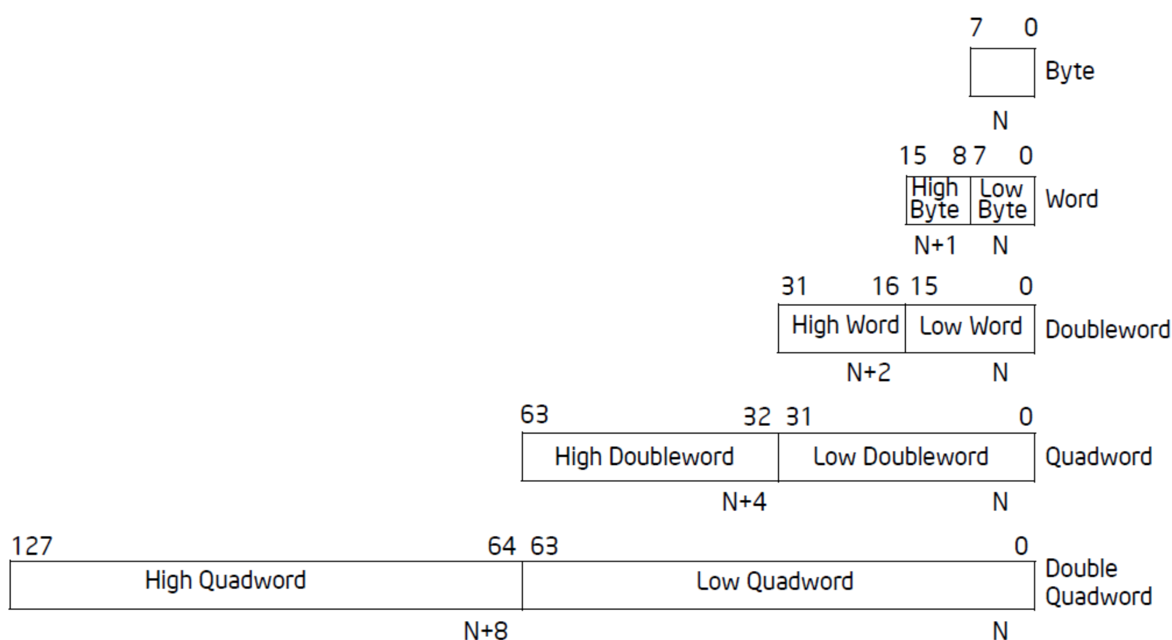


Рисунок 2.1 – Расположение информации в регистрах

Для долговременного хранения информации используется оперативная память. Байт является минимально адресуемой единицей информации в оперативной памяти.

При этом следует учитывать порядок расположения байт в регистрах и хранение этих байт в памяти. Выделяют два порядка хранения байт в памяти:

- Порядок от старшего к младшему (big-endian);
- Порядок от младшего к старшему (little-endian).

Порядок от старшего к младшему или big-endian: запись начинается со старшего и заканчивается младшим байтом. Этот порядок является стандартным для протоколов TCP/IP (поэтому его часто называют сетевым порядком байтов) и используется процессорами IBM 360/370/390, Motorola 68000, SPARC (рисунок 2.2, a). В этом же виде (используя представление в десятичной системе счисления) записываются числа индийско-арабскими цифрами в письменностях с порядком знаков слева направо.

Порядок от младшего к старшему или little-endian: запись начинается с младшего и заканчивается старшим байтом. Этот порядок записи принят в памяти персональных компьютеров с x86-процессорами, в связи с чем иногда его называют интеловский порядок байтов (рисунок 2.2, b).

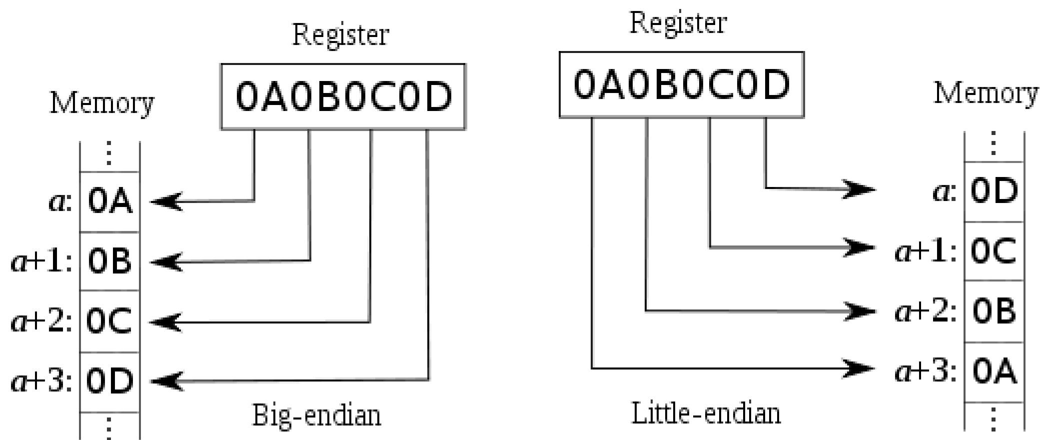


Рисунок 2.2 – Различия порядка байт big-endian и little-endian

Примеры записи чисел в памяти представлены на рисунке 2.3.

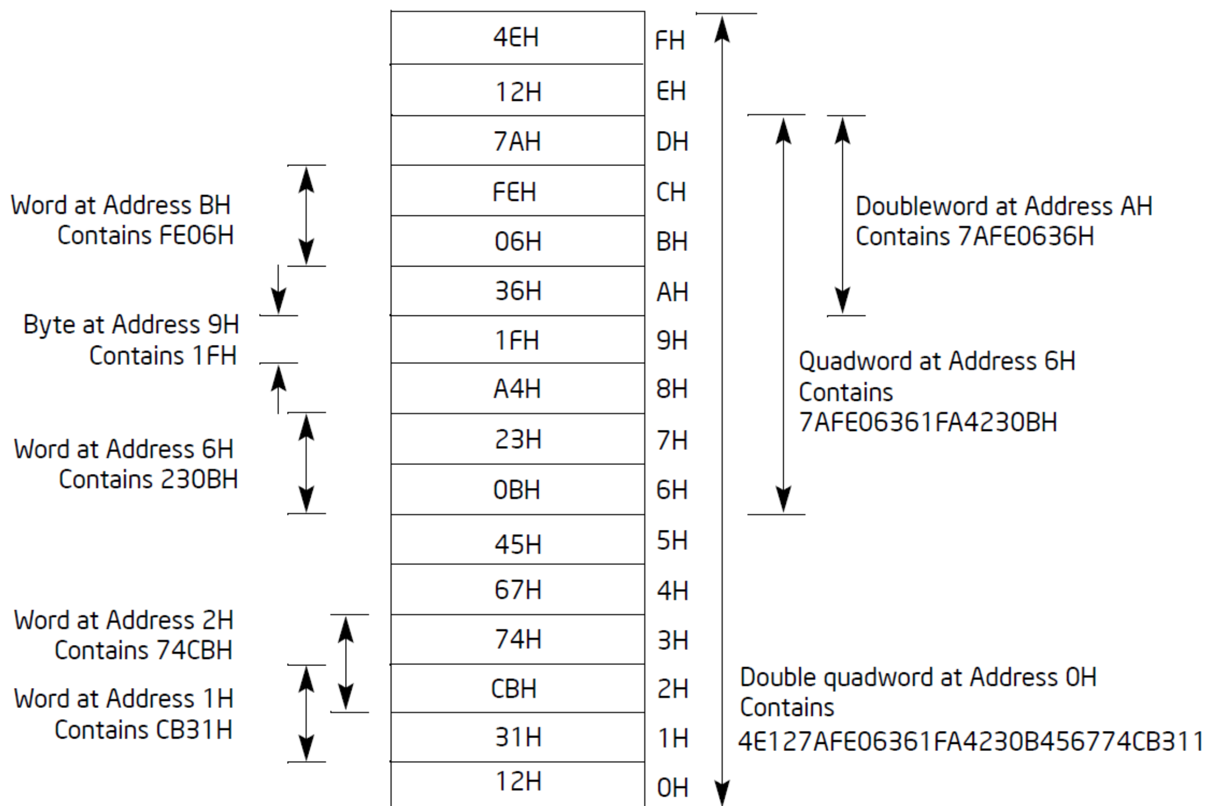


Рисунок 2.3 – Примеры записи чисел в памяти

2.2 Пользовательские регистры

Как следует из названия, *пользовательскими* регистры называются потому, что программист может использовать их при написании своих программ (рисунок 2.4). К этим регистрам относятся:

- восемь 32-битных регистров, которые могут использоваться программистами для хранения данных и адресов (их еще называют регистрами общего назначения (РОН)) (рисунок 2.5);

- шесть регистров сегментов;

- регистры состояния и управления (регистр флагов и регистр указателя команды).

На рисунке 2.4 некоторые регистры разделены, но это не разные регистры – это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты.

Так сделано для обеспечения работоспособности программ, написанных для младших 16-разрядных моделей микропроцессоров фирмы Intel, начиная с i8086.

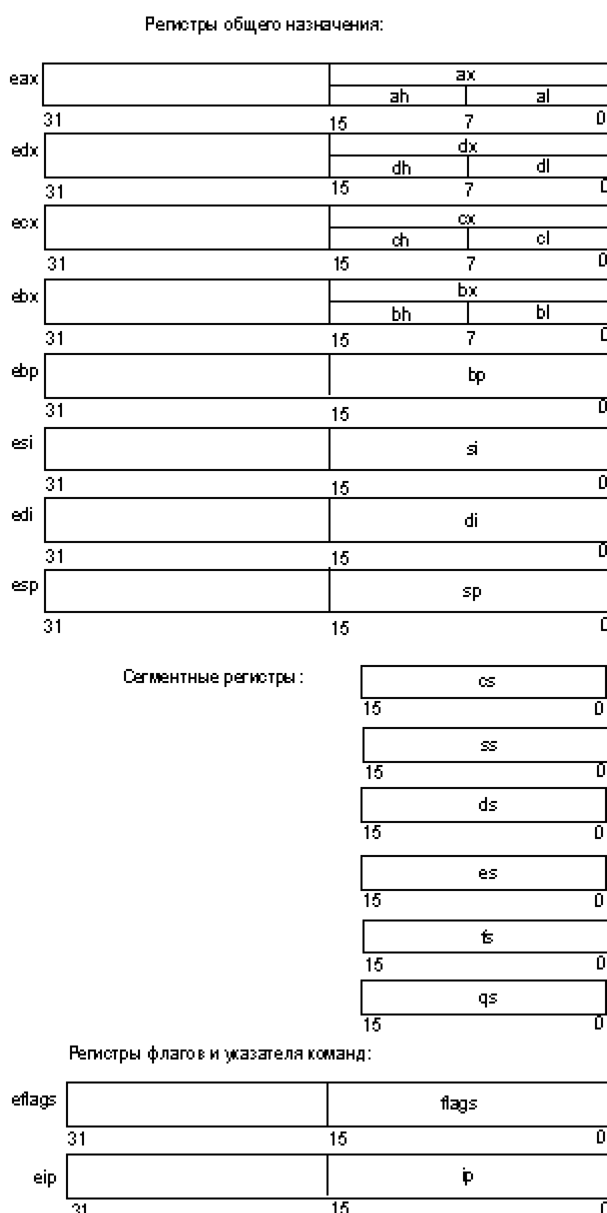


Рисунок 2.4 – Пользовательские регистры микропроцессоров x86

Микропроцессоры i486, Pentium и последующие модели имеют в основном 32-разрядные регистры. Их количество, за исключением сегментных регистров, такое же, как и у i8086, но размерность больше, что и отражено в их обозначениях — они имеют приставку **e** (*Extended*) .

General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Рисунок 2.5 – Регистры общего назначения

Разберемся подробнее с составом и назначением пользовательских регистров.

2.2.1 Сегментные регистры

В программной модели микропроцессора имеется шесть сегментных регистров: *CS, SS, DS, ES, GS, FS*. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых *сегментами*. Соответственно, такая организация памяти называется ***сегментной***.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены *сегментные регистры*. В этих регистрах содержатся адреса памяти, с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

Сегмент кода. Содержит команды программы. Для доступа к этому сегменту служит регистр **CS** (code segment register) — *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор (то есть эти команды загружаются в конвейер микропроцессора).

Сегмент данных. Содержит обрабатываемые программой данные. Для доступа к этому сегменту служит регистр **DS** (data segment register) —

сегментный регистр данных, который хранит адрес сегмента данных текущей программы.

Сегмент стека. Этот сегмент представляет собой область памяти, называемую *стеком*. Работу со стеком микропроцессор организует по следующему принципу: *последний записанный в эту область элемент выбирается первым*. Для доступа к этому сегменту служит регистр **SS** (stack segment register) — *сегментный регистр стека*, содержащий адрес сегмента стека.

Дополнительный сегмент данных. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре *DS*. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре *DS*, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных *префиксов переопределения сегментов* в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах **ES, GS, FS** (extension data segment registers).

Однако при работе в ОС Windows (работающая в защищенном режиме) используется несколько другая модель памяти. В ней сегментные регистры содержат так называемые селекторы, указывающие на единый сегмент данных. В этом случае регистры не должны изменяться прикладными программами непосредственно.

2.2.2 Регистры общего назначения (РОНы)

Все регистры этой группы позволяют обращаться к своим “младшим” частям. Использовать для самостоятельной адресации можно только младшие 16 и 8-битные части этих регистров. Старшие 16 бит этих регистров как самостоятельные объекты недоступны. Это сделано для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel.

Регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства, то их еще называют *регистрами АЛУ*:

EAX/AX/AH/AL (Accumulator register) — *аккумулятор*. Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;

EBX/BX/BH/BL (Base register) — *базовый* регистр. Применяется для хранения базового адреса некоторого объекта в памяти;

ECX/CX/CH/CL (Count register) — *регистр-счетчик*. Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы

соответствующей команды. К примеру, команда организации цикла **loop** кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра *ECX/CX*;

EDX/DX/DH/DL (Data register) — регистр *данных*. Так же, как и регистр *EAX/AX/AH/AL*, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

Следующие два регистра (индексные регистры) используются для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

ESI/SI (Source Index register) — *индекс источника*. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;

EDI/DI (Destination Index register) — *индекс приемника* (получателя).Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как ***стек***. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

ESP/SP (Stack Pointer register) — регистр *указателя стека*. Содержит указатель вершины стека в текущем сегменте стека.

EBP/BP (Base Pointer register) — регистр *указателя базы кадра стека*. Предназначен для организации произвольного доступа к данным внутри стека.

Большинство из них могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать. Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. Знание этих особенностей позволит вам при необходимости хотя бы на несколько байт сэкономить память, занимаемую кодом программы.

2.2.3 Регистры состояния и управления

В микропроцессор включены несколько регистров, которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

- регистр флагов **EFLAGS/FLAGS** (рисунок 2.6);
- регистр указателя команды **EIP/IP**.

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора.

EFLAGS/FLAGS (flag register) — регистр *флагов*. Разрядность *EFLAGS/FLAGS* — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру *FLAGS* для i8086.

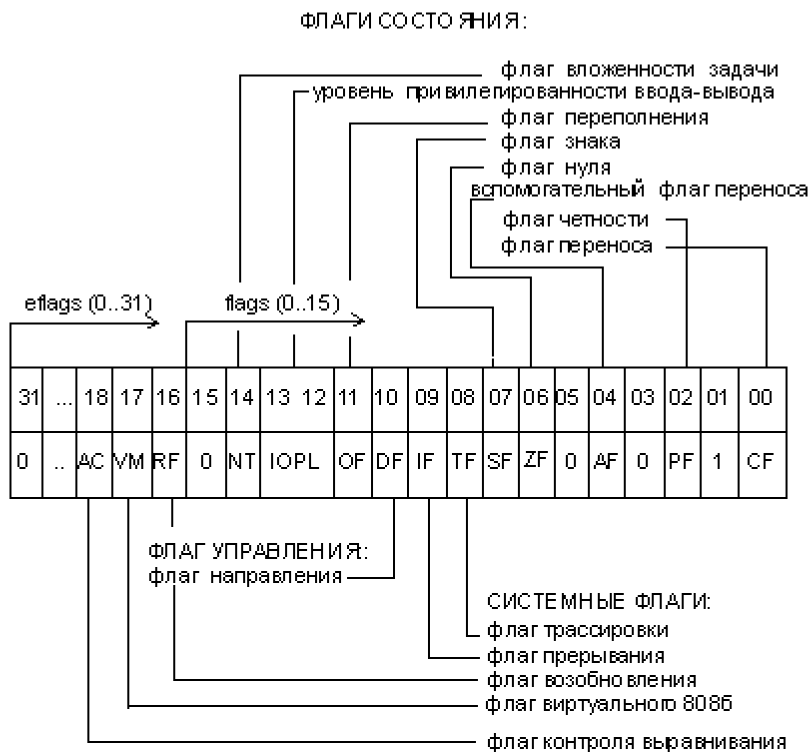


Рисунок 2.6 – Содержимое регистра EFLAGS

Исходя из особенностей использования, флаги регистра *EFLAGS/FLAGS* можно разделить на три группы:

8 флагов состояния. Эти флаги могут изменяться после выполнения машинных команд. *Флаги состояния* регистра *EFLAGS* отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.

1 флаг управления. Обозначается **DF** (Directory Flag). Он находится в 10-м бите регистра *eflags* и используется цепочечными командами. Значение флага *DF* определяет направление поэлементной обработки: от начала строки к концу (*DF* = 0) или, наоборот, от конца строки к ее началу (*DF* = 1). Для работы с флагом *DF* существуют специальные команды: cld (снять флаг *DF*) и std (установить флаг *DF*). Применение этих команд позволяет привести флаг *DF* в соответствие с алгоритмом и обеспечить

автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;

5 системных флагов, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086.

Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

2.3 Типы данных и описание данных

Основной тип данных микропроцессоров x86 – целый 32-х разрядный, возможна также обработка “наследственных” 16-ти и 8-ми разрядных данных. В 16-х разрядных микропроцессорах семейства x86 основной тип данных – целый 16-разрядный, но возможно использование также 8-разрядных целых.

Обычно данные в программе на языке ассемблера описываются в сегменте данных, но учитывая, что встроенный ассемблер не поддерживает описание данных, в практических занятийах будем использовать объявления переменных языка.

При описании данных и их типов (размеров) используются директивы ассемблера, указанные в таблице 2.1.

Таблица 2.1 – Основные типы данных

Размер	**Директива ассемблера**	**Соответствующей тип C++ (32-х битное приложение)**
Байт (8 бит)	db	char
Слово (16 бит)	dw	short
Двойное слово (32 бита)	dd	int/long/float
Учетверенное слово (64 бита)	dq	double/long double
10 байт (80 бит)	dt	-

2.4 Команды пересылки и загрузки

Для передачи данных между различными ячейками памяти используется следующие команды.

2.4.1 mov (Переслать)

MOV «Переслать» (MOVe) – пересылает байт, слово или двойное слово из операнда источника в операнд назначения.

```
mov D, S
```

* Здесь и далее `D` – приемник, `S` – источник, `O` – операнд.

Перемещает данные из источника в назначение. При этом разрядность операндов должна совпадать. Нельзя переписывать память в память и данные из одного сегментного регистра в другой и в сегментный регистр непосредственное значение.

Допустимые варианты формата команды:

```
mov R, IM     ; непосредственные данные в регистр
mov R, R      ; между регистрами общего назначения
mov SR, R     ; из РОНа в системный регистр
mov M, IM     ; непосредственные данные в память
mov R, M      ; из памяти в регистр
mov R, SR     ; из системного регистра в РОН
mov M, R      ; из регистра в память
```

* Здесь и далее `R` – регистр (общего назначения), `M` – ячейка памяти, `IM` – непосредственное значение (символьная константа), `SR` – системный регистр.

2.4.2. xchg (Обмен)

XCHG «Обмен (Перестановка)» (eXCHaGe) – переставляет содержимое двух операндов.

```
xchg O1, O2
```

Эта команда используется вместо трех команд mov. Она не требует временного размещения в памяти одного из операндов в то время, когда другой загружается. Команда xchg особенно полезна при использовании семафоров или аналогичных структур данных в процессе синхронизации.

Команда xchg может менять местами два байта, два слова или два двойных слова. Операндами для команды xchg могут служить два регистровых операнда или операнд-регистр и операнд, расположенный в памяти. Запрещается использовать m, m.

2.4.3 lea (Загрузить эффективный адрес)

LEA «Загрузить эффективный адрес» (Load Effective Address) – помещает 32-разрядный сдвиг операнда-источника в памяти (а не его содержимое) в операнд назначения.

```
lea D, S
```

Операнд-источник должен находиться в памяти, и операнд назначения должен быть регистром общего назначения. Эта команда особенно полезна для инициализации регистров ESI и EDI перед выполнением команд работы со строками.

Загрузка эффективного адреса (адрес смещения относительно начала сегмента, в котором находится переменная (метка) M). В отличие от mov помещает адрес смещения, а не значения.

```
lea R, M          ; данные две записи эквивалентны
mov R, offset M
```

Команда lea может выполнять любое необходимое индексирование или масштабирование.

```
lea EBX, EBCDIC_TABLE
```

Заставляет процессор поместить адрес начальной позиции таблицы, помеченной как EBCDIC_TABLE, в EBX.

2.4.4 Команды загрузки (пересылки) флагов

Хотя и существуют специальные команды для изменения флагов CF и DF, нет прямых методов изменения для остальных флагов, ориентированных на использование в прикладном программировании. Команды пересылки флага позволяют программе изменять состояние других флагов с использованием команд манипуляций с битами, если только эти флаги были перемещены в стек или в регистр AH.

LAHF «Загрузить в регистр AH из флагов» (Load into AH register from Flags) – копирует содержимое младших 8 бит регистра флагов в регистр AH (копируются биты 7,6,4,2 и 0, содержимое оставшихся битов 5,3, и 1 остается неопределенным). Содержимое регистра EFLAGS остается неизменным.

SAHF «Сохранить значения регистра AH во флагах» (Store AH into Flags) копирует содержимое регистра AH в младшие 8 бит регистра флагов (копируются биты 7, 6, 4, 2, и 0 во флаги SF, ZF, AF, PF и CF соответственно).

PUSHF/PUSHFD «Поместить флаги в стек» (PUSH Flags) – сохраняет младшее слово регистра FLAGS/EFLAGS в стеке.

POPF/POPFD «Извлечь флаги из стека» (POP Flags) восстанавливает слово из стека в регистр FLAGS/EFLAGS.

2.4.5 Команды работы со стеком

PUSH «Поместить» - копирует операнд-источник в вершину стека (декрементируя перед этим указатель стека (регистр ESP)).

```
push S
```

Команда PUSH часто используется для размещения в стеке параметров перед вызовом процедуры. Команда PUSH работает с операндами, размещенными в памяти, непосредственными операндами и с регистровыми операндами (включая регистры сегмента).

POP «Восстановить (данные) из стека» передает слово или двойное слово из текущей вершины стека (на которую указывает регистр ESP) операнду назначения и затем увеличивает значение регистра ESP, чтобы тот

указывал на новую вершину стека. POP перемещает информацию из стека в регистр общего назначения, регистр сегмента или в память.

```
pop D
```

PUSHA/PUSHAD «Поместить (в стек) все 16/32 разрядные регистры в стек» (PUSH All) – сохраняет содержимое восьми регистров общего назначения в стеке. Эта команда упрощает вызовы процедур путем сокращения числа команд, необходимых для сохранения содержимого регистров общего назначения. Процессор размещает регистры общего назначения в стеке в следующем порядке: EAX, ECX, EDX, EBX, начальное значение регистра ESP перед тем, как был размещен регистр EAX, EBP, ESI и EDI. Результат выполнения команды PUSHA противоположен действию команды POPA. Вызывается без параметров.

```
pusha
```

POPA/POPAD «Восстановить (из стека) все регистры» (POP All) – восстанавливает из стека данные, сохраненные в нем при помощи команды PUSHA, в регистры общего назначения, за исключением регистра ESP. Значение регистра ESP восстанавливается после выполнения чтения стека (Восстановления). Вызывается без параметров.

```
popad
```

2.5 Режимы адресации

В зависимости от того откуда и куда передаются данные и каким способом, различаются несколько режимов адресации:

1) регистровая;
2) непосредственная;
3) прямая;
4) косвенная;
5) по базе со смещением;
6) по базе с индексированием;
7) стековая.

2.5.1 Регистровая адресация

Этот вид адресации предполагает передачу данных из одного регистра в другой.

Пример 2.1 Регистровая адресация

```
mov EAX, EBX ; копируются данные из EBX в EAX

mov DX, BX   ; копируются данные из BX в DX
mov AL, CH   ; копируются данные из CH в AL
```

```
xchg AL, AH  ; обмен значений регистров AL и AH
```

2.5.2 Непосредственная адресация

При этом виде адресации происходит загрузка в регистр константы непосредственно записанной в команде.

Пример 2.2 Непосредственная адресация

```
; счетчик проинициализировать числом 10
mov ECX, 10
; Поместить в EAX число 10h (16)
mov EAX, 10h
; загрузка адресов в индексный регистр
lea ESI, X
; загрузка адресов в регистр EBX
lea EBX, Y
```

2.5.3 Прямая адресация

Здесь предполагается передача данных из ячейки памяти в другую ячейку памяти, или из ячейки памяти в регистр, или из регистра в ячейку памяти.

Пример 2.3 Прямая адресация

```
; скопировать в BX слово по адресу ES:00404000
; (Необходимо иметь доступ к данным, расположеннм по этому адресу!
; В противном сучае обращение вызовет исключение.)
mov BX, ES:00404000h
; Если в сегменте, на который указывает DS, есть переменная с именем X
; размером двойное слово, то значение хранящееся в EDX можно сохранить
; в X так:
mov [X], EDX
; или так:
mov X, EDX
; Поместить данные из переменной в регистр:
mov EAX, Z
```

2.5.4 Косвенная адресация

Этот вид адресации предполагает использование базового регистра BX/EBX, используемого в качестве указателя.

Пример 2.4 Косвенная адресация

```
; пусть в сегменте, на который указывает DS, первым объявлено слово p,
; а следом за ним - слово q; тогда их значения можно скопировать так:
lea EBX, [p]   ; Получаем эффективный адрес переменной
mov AX, [EBX]  ; Копируем данные в регистр из яейки с адресом равным EBX
add EBX, 000000002h ; Перемещаемся к следующей ячейке (к переменной q)
mov CX, [EBX]  ; Копируем данные в регистр из яейки с адресом равным EBX
```

2.5.5 Адресация по базе со смещением

Этот вид адресации представляет из себя разновидность косвенной адресации.

Пример 2.5 Адресация по базе со смещением

```
; следующие команды из примера 4 можно заменить на одну:
; lea EBX, [p]
; add EBX, 000000002h
; mov CX, [EBX]
mov CX, [EBX + 000000002h]
```

2.5.6 Адресация по базе с индексированием

Этот вид адресации представляет из себя разновидность косвенной адресации и предполагает использование индексных регистров SI/ESI и DI/EDI.

Пример 2.6 Адресация по базе с индексированием

```
; Пусть объявлен массив слов array; тогда получить значения
; первого, третьего и пятого элемента можно так:
mov EBX, offset array
mov ESI, 00000000h
mov AX, [EBX][ESI] ; 1
add ESI, 4
mov AX, [EBX][ESI] ; 3
add ESI, 4
mov AX, [EBX][ESI] ; 5
```

2.5.7 Стековая адресация

Стековая адресация предназначена для передачи данных из стека и в стек. Для этого предназначены команды push и pop, адресующие ячейки памяти посредством регистра SP, и команды mov, использующие регистр BP/EBP.

Пример 2.7 Временное хранение данных

```
push    EAX     ; сохраняем текущее значение EAX в стеке
...             ; выполняем какие нибудь действия,
...             ; использующие EAX
pop     EAX     ; восстанавливаем прежнее значение EAX
```

Пример 2.8 Копирование содержимого одного сегментного регистра в другой

```
push    DS
pop     ES
```

Пример 2.9 Выделение старшей части расширенного регистра данных (с помещением значения в другой регистр)

```
push    EAX     ; помещаем значение eax в стек (4 байта)
```

```
pop     AX      ; извлекаем младшую часть (2 байта)
pop     BX      ; извлекаем старшую часть (2 байта)
```

Пример 2.10 Извлечение данных из произвольной ячейки стека

```
mov EBP, ESP
mov AX, [EBP + 2]
```

В общем виде общую схему адресации (объединяющую все виды адресации) можно представить как сумму следующих значений (рисунок 2.7):

Эффективный адрес = База + (Масштаб*Индекс) + Смещение

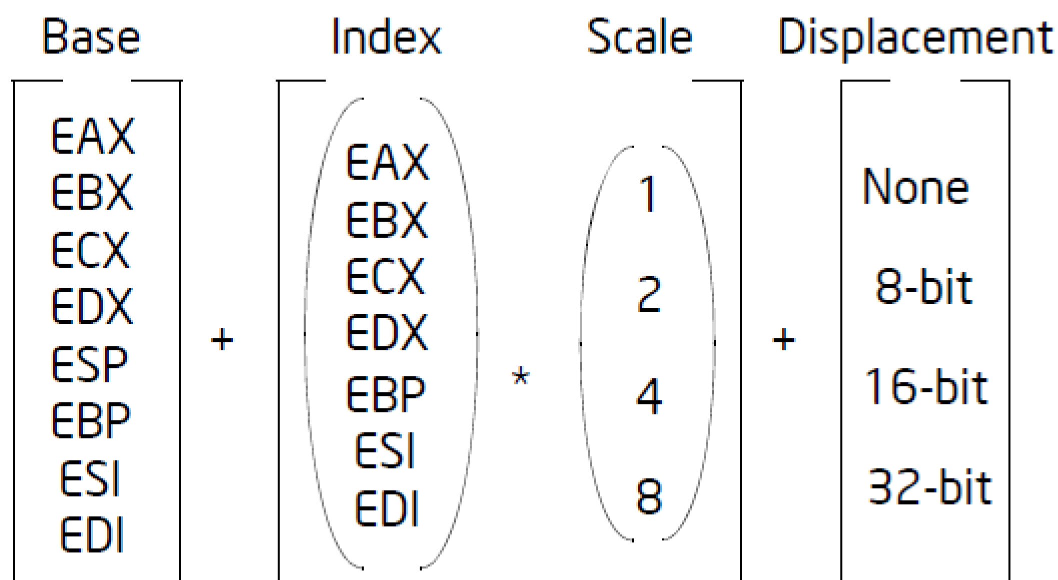


Рисунок 2.7 – Схема получения эффективного адреса

2.6 Флаговые операции

Иногда необходимо в процессе выполнения программы изменять состояние процессора. Для этого предназначены команды работы с флагами.

Пример 2.11 Работа с флагами

```
lahf ; загружаем в ah содержимое флагового регистра
stc  ; устанавливаем флаг cf
clc  ; сбрасываем флаг cf
cmc  ; инвертируем флаг cf
std  ; устанавливаем флаг df
cld  ; сбрасываем флаг df
cli  ; сбрасываем флаг if
sti  ; устанавливаем флаг if
sahf ; восстанавливаем из ah флаговый регистр
```

3 Контрольные вопросы

1) Опишите регистры процессоров семейства x86?
2) Опишите структуру регистра флагов?
3) Опишите форматы и алгоритмы работы следующих команд: mov, lea, lds, les, lfs, lgs, lss, xchg?
4) Форматы и алгоритмы работы команд работы с флагами: lahf, sahf, clc/stc, cld/std, cmc, cli/sti?
5) Как поместить в регистр адрес переменной, а не ее значение?
6) Как получить содержимое ячейки зная ее сегмент и смещение?
7) Режимы адресации, применяемые в процессорах семейства x86.
8) Опишите режим регистровой адресации.
9) Опишите режим непосредственной адресации.
10) Опишите режим прямой адресации.
11) Опишите режим косвенной адресации.
12) Опишите режим адресации по базе со сдвигом.
13) Опишите режим адресации по базе с индексированием.
14) Общее понятие стека.
15) Особенности реализации стека в семействе микропроцессоров x86.
16) Форматы, предназначение и алгоритмы работы команд стека: push, pop, pusha/pushad, popa, popad.
17) Примеры использования команд работы со стеком.

4 Задание

1) Протестировать все примеры, приведенные в разделе "Краткая теория".
2) Оформить отчёт.

Практическое занятие № 3. Арифметические команды целочисленного устройства микропроцессора

1 Цель работы

Исследовать с помощью отладчика работу арифметических команд. Научиться использовать арифметические команды целочисленного устройства для вычисления простых выражений.

2 Краткая теория

Арифметико-логическое устройство микропроцессора поддерживает две категории арифметических команд:
- команды двоичной целочисленной арифметики;
- команды двоично-десятичной арифметики.

2.1 Представление данных в микропроцессоре

На аппаратном уровне микропроцессор выполняет арифметические операции над данными, представленными двоичным кодом.

Соответственно при выполнении операций результаты представляются в двоичной системе счисления. Однако при этом возникает проблема представления отрицательных чисел. Существует две формы представления отрицательных чисел: обратный код и дополнительный код.

В МП x86 для этих целей используется дополнительный код. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ. Дополнительный код отрицательного числа можно получить инвертированием модуля двоичного числа и прибавлением к инверсии единицы, либо вычитанием числа из нуля.

При записи числа в дополнительном коде старший разряд является *знаковым*. Если его значение равно 0, то в остальных разрядах записано положительное двоичное число, совпадающее с прямым кодом. Если число, записанное в прямом коде, отрицательное, то все разряды числа инвертируются, а к результату прибавляется 1. К получившемуся числу дописывается старший (знаковый) разряд, равный 1.

Двоичное 8-разрядное число со знаком в дополнительном коде может представлять любое целое в диапазоне от −128 до +127. Если старший разряд равен нулю, то наибольшее целое число, которое может быть записано в оставшихся 7 разрядах равно 2^7-1, что равно 127.

Если двоичные числа имеют положительные значения, это определяется нулевым значением старшего (самого левого) разряда. Отрицательные двоичные числа, наоборот, имеют установленный в единицу бит в старшем разряде и записаны в дополнительном коде.

Для представления отрицательного двоичного числа необходимо инвертировать все биты и прибавить 1.

Пример 3.1 Преобразование 8-битного числа в дополнительный коду

```
Исходное число 42 :  00101010b
Инвертирвоание    :  11010101b
Добавление 1      :  11010110b (Число -42 в дополнительном коде)
```

Компилятор языка ассемблера выполняет это преобразование автоматически на этапе ассемблирования программы, поэтому для объявления отрицательной переменной достаточно просто указать знак перед значением.

```
x dd -5 ; В ячейке памяти будет размещено занчение 0xFFFFFFFB
```

Однако при интерпретации результатов необходимо учитывать, что с точки зрения микропроцессора не существует как такового отрицательного или положительного числа, а имеется набор бит размещенных в регистре. Поэтому в зависимости от используемых команд они могут быть интерпретированы и как число со знаком и как число без знака.

Изначально ассемблер не имеет встроенных средств для контроля знаковых/беззнаковых типов, поэтому данная задача ложится на программиста.

```
x dd -5 ; Размещенное занчение 0xFFFFFFFB можно интерпретирвоать как
        ; беззнаковое 32-х битное число 4294967291
```

Арифметика чисел в дополнительном коде сводится к замене вычитания на сложение с числом в дополнительном коде. При этом инвертируется знак вычитаемого, которое затем складывается с уменьшаемым.

```
Уменьшаемое                ( 77) : 01001101b
Вычитаемое                 ( 42) : 00101010b
Перобразованное вычитаемое (-42) : 11010110b


Уменьшаемое (77) / Слагоемое 1 ( 77)      :    01001101b
Вычитаемое  (42) / Слагоемое 2 (-42)      :    11010110b
Разность (77-42) / Сумма (77+(-42))  = 35 : (1)00100011b
```

В результате сложения теряется старший бит, что, в итоге дает корректный результат вычитания/сложения с отрицательным числом.

Для определения абсолютного значения отрицательного двоичного числа, необходимо выполнить обратное преобразование, повторив операции инвертирования всех битов и добавления 1.

Пример 3.2. Преобразование числа к дополнительному коду

```
Исходное число -42 :  11010110b
Инвертирвоание     :  00101001b
Добавление 1       :  00101010b (Исходное число 42)
```

Соответственно сумма +42 и -42 должна быть равна нулю.

Пример 3.3. Проверка сложения чисел в дополнительном коде

```
Первое число  42   :    11010110b
Второе число -42   :    00101010b
Результат сложения : (1)00000000b
```

Все восемь бит имеют нулевое значение. А не поместившийся единичный бит, вытесняемый влево, теряется.

Важным моментом при работе с арифметическими операциями является отслеживание переполнения результатов вычислений. Поскольку диапазон значений, который может храниться в ячейках различной емкости ограничен, то возможна ситуация при которой результат выполнения операции не помещается в приемник. В этом случае микропроцессор сообщает об этом путем установки флага переноса (CF).

Рассмотрим переполнение на примере регистра AL.

Пример 3.4. Переполнение регистров

```
mov AL, 254 ; AL = 254 (0xFE)
inc AL      ; AL = 255 (0xFF)
inc AL      ; AL = 0   (0x00) (0x(1)00)
;                              ^-переполнение – старший бит теряется!
;
mov AL, 250 ; AL = 250 (   0xFA)
add AL, 10  ; AL =  10 (   0x0A)
; Результат:; AL =   4 (0x(1)04)
;                         ^-переполнение – старший бит теряется!
```

Чтобы учесть этот эффект используется специальные команды, учитывающие потерю (перенос) значащего бита.

Ряд двоичных чисел переходящих через ноль выглядит следующим образом:

```
+3  00000011b
+2  00000010b
+1  00000001b
 0  00000000b
-1  11111111b
-2  11111110b
-3  11111101b
```

Фактически нулевые биты в отрицательном двоичном числе определяют его величину: позиционные значения нулевых битов можно рассматривать как единицы у положительных.

2.2 Основные команды

В язык ассемблера входит относительно небольшое количество примитивных математических операторов. Математические команды

ограничиваются сложением, умножением, делением и вычитанием знаковых и беззнаковых целых двоичных чисел.

- команды сложения add и adc; inc; xadd;
- команды вычитания sub и sbb; dec;
- команды умножения mul и imul;
- команды деления div и idiv;
- команда изменения знака neg.

2.2.1 Команды сложения

Команды сложения представлены четырьмя командами.

1. Сложение целых add помещает в приемник (заменяет) сумму значений источника и приемника.

```
add DST, SRC
```

Команда оказывает воздействие на флаги OF, SF, ZF, AF, PF и CF. Арифметические команды работают с 8-, 16-, или 32-разрядными данными. Флаги обновляются в зависимости от размеров операций. Например, 8-разрядная команда add устанавливает флаг CF равным 1, если сумма операндов превышает 255 (десятичное).

2. Сложение целых с переносом adc помещает в приёмник сумму значений источника и приёмника плюс 1, если флаг CF установлен. Если флаг CF сброшен, команда adc выполняет те же действия, что и команда add.

```
adc DST, SRC
```

Команда adc используется для поддержки переноса, когда сложение выполняется на одном из этапов, например, когда 32-разрядная команда add для сложения двух учетверенных слов-операндов. Команда оказывает воздействие на флаги OF, SF, ZF, AF, PF, и CF.

По сути, команда выполняет следующую операцию:

```
DST <= DST + SRC + CF
```

3. Инкремент inc добавляет 1 к операнду назначения.

```
inc DST
```

Команда inc сохраняет значение флага CF. Это позволяет использовать команду inc для обновления счетчиков циклов без оказания воздействия на флаги состояния, изменяющиеся под воздействием арифметических операций, используемых при управлении циклом. Флаг ZF может быть использован для обнаружения ситуации возникновения переноса. Использование команды add с непосредственным значением 1 в качестве операнда выполняет увеличение с обновлением флага CF. Можно использовать однобайтную форму этой команды, когда операндом является

регистр общего назначения. Команда оказывает воздействие на флаги OF, SF, ZF, AF, и PF.

4. Сложение с обменом xadd выполняет обмен значений приемника и источника, а затем помещает сумму значений в приемник.

```
xadd DST, SRC
```

По сути, команда выполняет следующую операцию:

```
DST <=> SRC
DST <= DST + SRC
```

В итоге, в приёмнике оказывается сумма приёмника и источника, а в источнике – старое значение приемника.

Форматы команд сложения:

```
add reg/mem, imm       ; размер 8/16/32 бита
add reg, reg/mem       ; размер 8/16/32 бита
add mem/reg, reg       ; размер 8/16/32 бита
add reg/mem16, imm8    ; получатель 16 бит, источник 8 бит
add reg/mem32, imm8    ; получатель 32 бит, источник 8 бит

adc reg/mem, imm       ; размер 8/16/32 бита
adc reg, reg/mem       ; размер 8/16/32 бита
adc mem/reg, reg       ; размер 8/16/32 бита
adc reg/mem16, imm8    ; получатель 16 бит, источник 8 бит
adc reg/mem32, imm8    ; получатель 32 бит, источник 8 бит

inc reg/mem            ; размер 8/16/32 бита

xadd reg/mem, reg      ; размер 8/16/32 бита
```

2.2.2 Команды вычитания

Команды вычитания представлены тремя командами.

1. Вычитание sub отнимает источник от приёмника и помещает результат в приёмник на полученный результат.

```
sub DST, SRC
```

Если возникает заем единицы, устанавливается значение флага CF. Операндами могут быть байты, слова и двойные слова со знаком и без него. Команда оказывает воздействие на флаги OF, SF, ZF, AF, PF и CF.

2. Вычитание с заемом sbb вычитает операнд-источник из операнда назначения, заменяя значение преемника на результат вычитания минус 1, если установлен флаг CF. Если флаг CF очищен, команда sbb выполняет ту же операцию, что и команда sub.

```
sub DST, SRC
```

Команда SUB используется для поддержки заема разряда, когда выполняется вычитание чисел как один из этапов, например, когда используется 32-разрядная команда sub для вычитания одного учетверенного слова из другого. Команда оказывает воздействие на флаги OF, SF, ZF, AF, PF и CF.

По сути, команда выполняет следующую операцию:

```
DST <= DST - SRC - CF
```

По сути, команда выполняет следующую операцию:

3. Декремент dec вычитает 1 из операнда назначения.

```
dec DST
```

Команда dec сохраняет состояние флага CF. Это позволяет использовать команду dec для обновления счетчиков циклов без воздействия на состояние флагов, изменяемых под воздействием арифметических операций, используемых для управления циклом. Использование команды sub с непосредственным значением 1 в качестве операнда выполняет уменьшение, которое обновляет значение флага CF. Допустима однобайтная форма этой команды, когда операндом является регистр общего назначения. Команда оказывает воздействие на флаги OF, SF, ZF, AF, и PF.

Форматы команд вычитания:

```
sub reg/mem, imm      ; размер 8/16/32 бита
sub reg, reg/mem      ; размер 8/16/32 бита
sub mem/reg, reg      ; размер 8/16/32 бита
sub reg/mem16, imm8   ; получатель 16 бит, источник 8 бит
sub reg/mem32, imm8   ; получатель 32 бит, источник 8 бит

sbb reg/mem, imm      ; размер 8/16/32 бита
sbb reg, reg/mem      ; размер 8/16/32 бита
sbb mem/reg, reg      ; размер 8/16/32 бита
sbb reg/mem16, imm8   ; получатель 16 бит, источник 8 бит
sbb reg/mem32, imm8   ; получатель 32 бит, источник 8 бит

dec reg/mem           ; размер 8/16/32 бита
```

2.2.3 Команды умножения

Процессоры x86 разделяют команды умножения/деления для операндов со знаком и операндов без знака.

Так команда mul работает с целыми без знака, в то время как команда imul работает как с целыми со знаком, так и с целыми без знака.

1. Умножение целых без знака mul выполняет умножение без знака операнда-источника и регистра AL, AX или EAX.

```
mul SRC
```

Если источником является байт, процессор умножает его на значение, хранящееся в регистре AL и возвращает результат удвоенной длины в регистры AH и AL. Если исходный операнд является словом, процессор умножает его на значение, хранящееся в регистре AX и возвращает результат удвоенной длины в регистры DX и AX. Если операнд-источник является двойным словом, процессор умножает его на значение, хранящееся в регистре EAX и возвращает результат в виде учетверенного слова в регистры EDX и EAX. Команда MUL устанавливает флаги CF и OF, если старшая половина результата отлична от нуля; в противном случае флаги очищаются. Состояние флагов SF, ZF, AF и PF не определено.

Таблица 3.1 – Работа команды умножения

Источник	**Результат**
8 бит	AH:AL
16 бит	DX:AX
32 бита	EDX:EAX

Если источником является байт, процессор умножает его на значение, хранящееся в регистре AL и возвращает результат удвоенной длины в регистры AH и AL. Если исходный операнд является словом, процессор умножает его на значение, хранящееся в регистре AX и возвращает результат удвоенной длины в регистры DX и AX. Если операнд-источник является двойным словом, процессор умножает его на значение, хранящееся в регистре EAX и возвращает результат в виде учетверенного слова в регистры EDX и EAX. Команда MUL устанавливает флаги CF и OF, если старшая половина результата отлична от нуля; в противном случае флаги очищаются.

2 Умножение целых со знаком imul выполняет операцию умножения со знаком.

Команда IMUL имеет три формы:

```
imul SRC
imul DST, SRC
imul DST, SRC1, SRC2
```

1. Форма с одним операндом. Операнд может быть словом, байтом или двойным словом, расположенным в памяти или в регистре общего назначения. Эта команда использует регистры EAX и EDX как операнды по умолчанию тем же образом, что и команда mul.

2. Форма с двумя операндами. Одним из исходных операндов должен быть регистр общего назначения, другим операндом может быть как регистр общего назначения, так и ячейка памяти. Результат заменяет содержимое регистра общего назначения.

3. Форма с тремя операндами: два операнда являются источниками, третий операнд является преемником. Одним из операндов-источников является непосредственное значение, указанное в команде; вторым может

быть регистр общего назначения или ячейка памяти. Результат сохраняется в регистре общего назначения. Непосредственный операнд является целым со знаком в двоично-дополнительном коде. Если непосредственный операнд является байтом, процессор автоматически расширяет его со знаком до размера второго операнда, прежде чем выполнить умножение.

Во многих отношениях вышеприведенные три формы похожи:

- Длина результата равняется удвоенной длине операндов.

- Флаги CF и OF устанавливаются, когда значащие биты переносятся в старшую половину результата. Флаги CF и OF очищаются, когда старшая часть результата является продолжением знака младшей части результата. Состояние флагов SF, ZF, AF и PF неопределенно.

Тем не менее, форматы 2 и 3 отличаются, так как результат усекается до длины операндов, прежде чем он будет сохранен в регистре назначения. По причине такого усечения необходимо проверить флаг OF, чтобы быть уверенным, что ни один из значащих битов не потерян.

Форматы 2 и 3 команды imul также могут быть использованы с операндами без знака, так как вне зависимости от того, являются ли операнды целыми со знаком или без знака, младшая половина результата остается одной и той же. Однако флаги CF и OF не могут быть использованы, для того чтобы определить, отличается ли правая часть результата от 0.

Форматы команд умножения:

```
mul reg/mem                     ; размер 8/16/32 бита

imul reg/mem                    ; размер 8/16/32 бита
imul reg16, imm8                ; произведение 16 бит
imul reg16, imm16               ; произведение 16 бит
imul reg32, imm8                ; произведение 32 бита
imul reg32, imm32               ; произведение 32 бита
imul reg16, reg/mem16           ; произведение 16 бит
imul reg32, reg/mem32           ; произведение 32 бита
imul reg16, reg/mem16, imm8/16  ; произведение 16 бит
imul reg32, reg/mem32, imm8/32  ; произведение 32 бит
```

2.2.4 Команды деления

Процессор i486 подразделяет команды деления на операции деления с операндами со знаком и с операндами без знака.

Так команда div работает с целыми без знака, в то время как команда idiv работает как с целыми со знаком, так и с целыми без знака.

В обоих случаях генерируется исключение ошибки деления, если делитель равен 0 или частное слишком велико для регистров AL, AX и EAX.

1. Деление целых без знака div выполняет беззнаковое деление регистров AL, AX и EAX на операнд-источник.

```
div SRC
```

Делимое (аккумулятор) по длине в два раза больше, чем делитель (операнд-источник); частное и остаток имеют ту же длину, что и делитель. Нецелочисленные результаты усекаются в направлении 0. Остаток всегда меньше, чем делитель. Для деления байтов без знака наибольшим частным может быть число 255. Для деления слов без знака наибольшее частное равняется 62535. Для деления двойных слов наибольшее частное равно 2**32-1. Состояние флагов OF, SF, ZF, AF, PF и CF не определено.

2. Деление целых со знаком idiv выполняет деление со знаком сумматора на операнд-источник. Команда idiv использует те же регистры, что и команда div.

```
div SRC
```

Для деления байтов со знаком максимальное положительное частное равняется +127, минимальное отрицательное частное равняется -128. Для деления слов со знаком максимальное положительное частное равняется +32767, минимальное отрицательное частное равняется -32768. Для деления двойных слов со знаком максимальное положительное частное равняется +2**32-1, минимальное отрицательное частное равняется -2**31.

Нецелочисленные результаты усекаются в направлении 0. Остаток всегда имеет тот же знак, что и частное и меньше, чем делитель в выражении. Состояние флагов OF, SF, ZF, AF, PF и CF не определено.

Таблица 3.2 – Операнды команды деления

Размер операнда (Делитель)	**Делимое**	**Частное**	**Остаток**
Байт (8 бит)	AX	AL	AH
Слово (16 бит)	DX:AX	AX	DX
Двойное слово (32 бита)	EDX:EAX	EAX	EDX

Форматы команд деления:

```
idiv reg/mem   ; размер 8/16/32 бита

div reg/mem    ; размер 8/16/32 бита
```

2.2.5 Команда изменения знака

Команда инвертировать знак neg изменяет знак числа путем вычитания целого со знаком из нуля.

```
neg DST
```

Результатом работы команды neg является изменение знака операнда в двоичном дополнительном коде при сохранении его значения. Команда оказывает воздействие на флаги OF, SF, ZF, AF, PF и CF.

Формат команды изменения знака:

```
neg reg/mem      ; размер 8/16/32 бита
```

2.3 Операции с большими числами

Фиксированная ёмкость (количество бит) регистров накладывает ограничение на диапазон значений, с которыми естественным образом может оперировать микропроцессор. Так для 8-битных операндов оно ограничено 2**8 значениями, 16-битных – 2**16, 32-битных – 2**32 соответственно (для знаковых операндов диапазоны сокращаются вдвое).

Но потребности вычислений реальных программ не всегда могут быть удовлетворены указанными ограничениями. В этом случае используется так называемая «длинная арифметика», т.е. работа с операндами, разрядность (размер) которых превышает длину машинного слова (разрядность регистров процессора).

Начнем с операций сложения/вычитания. Все выводы, полученные для сложения, будут справедливы и для вычитания.

Суть сложения (вычитания) в длинной арифметике заключается в поразрядном выполнении операций с учетом переноса (заема).

Ситуация переноса (заема) возникает когда результат сложения (вычитания) не помещается в приёмнике и регистрируется флагом CF. Ситуация переполнения возникает, когда результат умножения не помещается в приёмнике. Она регистрируется при помощи флага OF.

Рассмотрим на примере работу сложения двух учетверённых слов (64-битные числа). Целиком эти значения не помещаются в регистры, однако над ними необходимо выполнить операцию сложения и сохранить результат в третьей переменной.

Пример 3.5. Сложение в длинной арифметике

```
; Объявляем переменные
x       dq 123456789ABCDEF0h
y       dq 23456789ABCDEF01h
; В памяти байты расположатся следующим образом:
; Смещение: 0  1  2  3  4  5  6  7
;       x = [F0|DE|BC|9A|78|56|34|12]
;       y = [01|EF|CD|AB|89|67|34|23]
z       dq ?

...

; При обращении не по байтам, по двойным словам группы байт
; считываются целиком и занимают соотвествующие позиции в регистре
; Модификатор dword ptr указывает, что единоразово требуется считать
; область памяти размером двойное слово

;      __ __ __ __
; x = [F0|DE|BC|9A|78|56|34|12] => EAX = [9A|BC|DE|F0]

mov EAX, dword ptr [x]  ; Помещаем младшую часть x в регистр EAX (0x9ABCDEF0)
;      __ __ __ __
```

```
; y = [01|EF|CD|AB|89|67|34|23] => EAX = [AB|CD|EF|01]
mov EBX, dword ptr [y]  ; Помещаем младшую часть y в регистр EBX (0xABCDEF01)
add EAX, EBX            ; Выполняем сложение.
                        ; Результат (0x1468ACDF1) не помещается в EAX,
                        ; что приводт к установке флага переноса CF
mov dword ptr [z], EAX  ; Сохраняем младшую часть в z
;
; z = [F1|CD|8A|46|??|??|??|??] <= EAX = [46|8A|CD|F1]

; Имя переменной по своей сути является указателем на начало области памяти,
; содержащей хранимое значение
; Добавляя к нему смещение мы получаем доступ к другим разрядам (частям) числа
;
; x = [F0|DE|BC|9A|78|56|34|12] => EAX = [12|34|56|78]
;                 ^
;         [x + 4] |

mov EAX, dword ptr [x + 4] ; Помещаем старшую часть x в EAX (0x12345678)
;
; y = [01|EF|CD|AB|89|67|34|23] => EAX = [23|45|67|89]
mov EBX, dword ptr [y + 4]  ; Помещаем старшую часть x в  EAX (0x23456789)
adc EAX, EBX                ; Выполняем сложение с учетом переноса
                            ;  предыдущей операции (0x3579BE02).
                            ; Итого x + y => 0x3579BE02468ACDF1
mov dword ptr [z + 4], EAX  ; Сохраняем старшую часть в z
;
; z = [F1|CD|8A|46|02|BE|79|35] <= EAX = [35|79|BE|02]
```

Данный алгоритм можно расширить для работы с числами произвольной длинны, организовав цикл, поочередно выполняющий требуемую операцию над соответствующими разрядами (частями) чисел.

Теперь перейдем к умножению.

Обычно команда умножения работает с операндами одинакового размера, не превышающего размеры самих регистров. Для умножения же больших чисел требуется выполнение некоторых дополнительных действий. Самый простой подход предполагает умножение каждого слова отдельно и сложение полученных результатов, как это реализуется при умножении «в столбик». Единственным отличием будет только то, что разрядность множителей будет соответствовать разрядности регистров, а не десятичных разрядов как при умножении «в столбик».

> *Примечание*. Помимо этого существуют более эффективные алгоритмы умножения больших чисел, например умножение Карацубы.

Для простоты рассмотрим умножение операндов размером слово с использованием 8-разрядных регистров. Результат такого умножения будет иметь размер не более чем двойное слово.

Схема умножения будет выглядеть следующим образом:

```
;        [A|B]
;        [C|D]
```

```
; ------------
;       [BxD]
;     [AxD]
;     [BxC]
;   [AxC]
; ------------
; [?|X|X|X|X]
```

* Здесь знак вопроса означает возможное переполнение результата после выполнения сложения.

Результат умножения каждой из пар байт не превосходит слово. Соответственно при умножении старших разрядов происходит «сдвиг» на соответствующее суммарное число позиций, как это делается при умножении «в столбик». При сложении частичных результатов умножения необходимо учитывать возможный перенос разряда.

Рассмотрим пример реализации такого умножения.

Пример 3.6. Длинное умножение «столбиком»

```
.586
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

.data
x       dw 1257h
y       dw 90A5h
z       dd 0

; Сегмент кода
.code
main:
    ; Схема алгоритма умножения в столбик
    ;       [A|B]
    ;       [C|D]
    ; ------------
    ;       [BxD]
    ;     [AxD]
    ;     [BxC]
    ;   [AxC]
    ; ------------
    ; [?|X|X|X|X]

    ; Очистим используемые все регистры
    xor EAX, EAX
```

```
xor EBX, EBX
xor EDX, EDX
xor ECX, ECX
xor ESI, ESI
xor EDI, EDI

mov AL, byte ptr [x]          ; x[0] -> AL    ( 57h )
mov AH, byte ptr [y]          ; y[0] -> AH    ( A5h )
mul AH                        ; AL*AH - >AX ( 3813h )
; Часть числа [??|??|38|13]
mov BX, AX                    ; AX -> BX

mov AL, byte ptr [x + 1]      ; x[1] -> AL    ( 12h )
mov AH, byte ptr [y]          ; y[0] -> AH    ( A5h )
mul AH                        ; AL*AH - >AX ( 039Ah )

mov CX, AX                    ; AX -> CX
; Часть числа [??|03|9A|??]

mov AL, byte ptr [x]          ; x[0] -> AL    ( 57h )
mov AH, byte ptr [y + 1]      ; y[1] -> AH    ( 90h )
mul AH                        ; AL*AH - >AX ( 30F0h )

mov SI, AX                    ; AX -> SI
; [??|30|F0|??]

mov AL, byte ptr [x + 1]      ; x[1] -> AL    ( 12h )
mov AH, byte ptr [y + 1]      ; y[1] -> AH    ( 90h )
mul AH                        ; AL*AH - >AX ( 0A20h )

mov DI, AX                    ; AX -> DI
; [0A|20|??|??]

; Суммируем части результата
; [??|??|38|13]     BX
; [??|03|9A|??]     CX
; [??|30|F0|??]     SI
; [0A|20|??|??]     DI
; -------------
; [0A|5C|C2|13]     z

add word ptr [z + 0], BX
adc word ptr [z + 1], CX
adc word ptr [z + 1], SI
adc word ptr [z + 2], DI

mov ECX, [z]

; Проверка результатов обычным умножением
xor EAX, EAX
xor EBX, EBX
xor EDX, EDX

mov AX, [x]
mov BX, [y]
```

```
    mul BX      ;0A5C:C213

    ; Выход из программы
quit:
    mov eax, 0
    invoke ExitProcess, eax
; Конец программы
end main
```

Примечание. Приведенный пример является демонстрацией идеи и не может служить образцом реализации. Он нерационально использует для хранения промежуточных результатов отдельные регистры и откладывает на конец этап сложения частей. Это выполняется только для демонстрации итогового суммирования. В обычной программе сложение необходимо выполнять сразу же после получения частичного произведения.

2.4 Пример целочисленных вычислений

Рассмотрим пример вычисления арифметического выражения $z(x,y) = -\frac{x^2 + xy - 2}{(y+1)^3}$, где x и y имеют размер двойное слово (для простоты предположим, что результат не может превышать значения 2**32).

Пример 3.7. Пример вычисления арифметического выражения

```
.586
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

; Сегмент данных
.data
x       dd 10
y       dd 3
z       dd ?

; Сегмент кода
.code
main:
    ; Вычисление z = -1*(x**2 + xy -2)/((y + 1)**3)
    ; Вычисляем x**2
    mov EBX, [x]    ; x -> EBX                  ( EBX =  10 = 0Ah )
    mov EAX, EBX    ; x -> EAX                  ( EAX =  10 = 0Ah )
    mul EAX         ; EAX = x**2                ( EAX = 100 = 64h )
```

```
    ; Сохраним промежуточные результаты в ECX
    mov ECX, EAX     ; x**2 -> ECX                 ( ECX = 100 = 64h )
    ; Вычисляем x*y
    mov EAX, EBX     ; x (уже хранимый в EBX) -> EAX ( EAX =  10 = 0Ah )
    mov EBX, [y]     ; y -> EBX                    ( EBX =   3 = 03h )
    mul EBX          ; EAX = x*y                   ( EAX =  30 = 1Eh )
    ; Вычитаем 2 из полученного произведения
    sub EAX, 2       ; EAX = x*x - 2               ( EAX =  28 = 1Ch )
    ; Добавляем x**2 к результату для получения числителя дроби
    add EAX, ECX     ; EAX = (x*y - 2) + x**2      ( EAX = 128 = 80h )

    ; Сохраним числитель в ECX
    mov ECX, EAX     ; (x*y - 2) + x^2 -> ECX      ( ECX = 128 = 80h )

    ; Вычисляем знаменатель дроби
    inc EBX          ; EBX = y + 1                 ( EBX =   4 = 04h )
    mov EAX, EBX     ; (y+1) -> EAX                ( EAX =   4 = 04h )
    mul EAX          ; EAX = (y+1)**2              ( EAX =  16 = 10h )
    mul EBX          ; EAX = (y+1)**2 * (y+1)      ( EAX =  64 = 40h )
    ; Приступаем к делению. Поместим знаменатель в EBX
    mov EBX, EAX     ; (y+1)**3 -> EBX             ( EBX =  64 = 40h )
    ; Поместим числитель в EAX
    mov EAX, ECX     ; (x*y - 2) + x**2 -> EAX     ( EAX = 128 = 80h )
    ; Обнулим старшую часть делимого (см. работу команды деления)
    mov EDX, 0       ; 0 -> EDX                    ( EDX =   0 = 00h )
    div EBX          ; EAX = EAX / EBX             ( EAX =   2 = 02h )
    ; Изменяем знак числа
    neg EAX          ; -EAX       ( EAX = -2 = FFFFFFFEh )
    ;
    mov z, EAX       ; EAX -> Z

    ; Выход из программы
quit:
    mov eax, 0
    invoke ExitProcess, eax
end main
```

2.5 Команды преобразования типов

Команды преобразования типа преобразуют байты в слова, слова в двойные слова и двойные слова в 64-разрядные значения (так называемые учетверенные слова). Эти команды особенно полезны при преобразовании целых со знаком, поскольку они автоматически заполняют дополнительные биты большего элемента значениями знакового бита меньшего элемента. Это приводит к целым того же знака и той же величины, только в более длинном формате. Такой тип преобразования называется распространением знака.

```
;|                              15                              0
;|  ДО РАСШИРЕНИЯ ЗНАКА           |S|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N
;|31                                                            0
;||S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|S|N|N|N|N|N|N|N|N|N|N|N|N|N|N|N
;  ПОСЛЕ РАСШИРЕНИЯ ЗНАКА
```

Имеется два вида команд преобразования типа:

1. Команды CWD, CBW и CWDE, которые работают только с данными в регистре EAX.

2. Команды MOVSX и MOVZX, которые позволяют одному из операндов быть регистром общего назначения, оставляя другому операнду возможность быть ячейкой памяти или регистром.

CWD (Преобразовать слово в двойное слово) и CDQ (Преобразовать двойное слово в учетверенное слово) удваивают размерность операнда-источника. Команда CWD копирует знак (бит 15) слова в регистре AX в каждый бит регистра DX. Команда CDQ копирует знак (бит 31) двойного слова в регистре EAX в каждый бит регистра EDX. Команда CWD может быть использована для получения делимого в формате двойного слова из слова перед началом деления слова, и команда CDQ может быть использована для получения делимого в формате учетверенного слова из двойного слова перед началом деления двойного слова.

1. Преобразовать байт в слово копирует знак (бит 7) байта в регистре AL в каждый бит регистра AX.

```
CBW
```

2. Преобразовать слово в двойное слово с расширением копирует знак (бит 15) слова в регистре AX в каждый бит регистра AX+ DX.

```
CWD
```

3. Преобразовать слово в двойное слово с расширением копирует знак (бит 15) слова в регистре AX в каждый бит регистра EAX.

```
CWDE
```

4. Преобразовать слово в двойное слово с расширением копирует знак (бит 15) слова в регистре AX в каждый бит регистра EAX + EDX.

```
CDQ
```

5. Переслать с распространением знака расширяет 8-разрядное значение до 16-разрядного значения или 8-разрядное или 16-разрядное значение до 32-разрядного значения, используя значение знакового бита для заполнения пустых позиций.

```
movsx DST, SRC
```

6. Переслать с расширением нулями расширяет 8-разрядное значение до 16-разрядного значения или 8-разрядное или 16-разрядное значение до 32-разрядного значения, очищая (заполняя нулями) пустые позиции.

```
movzx DST, SRC
```

3 Контрольные вопросы

1) Опишите принципы двоичной формы представления целых чисел.
2) Что такое дополнительный код?
3) Опишите процесс перевода чисел в дополнительный код.
4) Что такое переполнение?
5) Опишите команды целочисленного сложения.
6) Опишите команды целочисленного вычитания.
7) Опишите команды целочисленного умножения.
8) Опишите команды целочисленного деления.
9) Опишите работу команд преобразования типов.
10) Как изменить знак числа?
11) Опишите принцип работы длинной арифметики.
12) Опишите принцип работы длинной арифметики.
13) Каково назначение флага переноса CF?
14) Каково назначение флага переполнения OF?
15) Каково назначение флага знака SF?

4 Задание

1) Выполнить задание в соответствии с пунктом 5.1.
2) Выполнить задание в соответствии с пунктом 5.2.
3) Оформить отчёт.

5 Варианты заданий

5.1 Вычисление значения функции

Выполните 3 (три) упражнений из ниже приведенного списка, выбирая по следующему принципу: пусть номер студента в списке группы N, тогда выполняются упражнения с номерами N, N+5, N+7.

Написать программу, вычисляющую значение z для заданных x и y.

1. z = (x**2 + 2*y - 45) / (x**3);
2. z = 1 / y + x**3 - 32;
3. z = (3 + x/y) / (x-y+1);
4. z = x / (x - y + x*y);
5. z = (4 - (x+3)/(y-1))*(-xy);
6. z = ((x+1)/y - 1)*2x;
7. z = y*(2-(y+1)/x);
8. z = (xy - 1)/(x+y);
9. z = x**3 + y -1;
10. z = (xy + 1)/ x**2;
11. z = (x+y)/(x-y);
12. z = - 1/x**3 + 3;
13. z = x - y/x +1;
14. z = ((x+y)/y**2 - 1)*x;

15. z = (x-y)/(xy+1);
16. z = - x/y+y**2 +3;
17. z = y**2 + xy + x/y;
18. z = (1 + x * y)/2;
19. z = -(1-y)/(1+x);
20. z = -x*(1-xy);
21. z = y+x/y-1;
22. z = 5/xy+x**3;
23. z = -x + y**3 - 1;
24. z = x**3 / (x-y);
25. z = x**3 -2x**2*y+1;
26. z = -3x + y**2 +1;
27. z = -(x/y +1)/y**2;
28. z = 1+x**2/3y;
29. z = y-x/3+1;
30. z = (xy)**3 +1/y.

5.2 Длинная арифметика

Реализовать программу вычисляющую значение функции $f(x,y,z)$ и работающую с учетверенными словами.

1) $f(x,y,z) = x*y - z$
2) $f(x,y,z) = x + y - z$
3) $f(x,y,z) = x - y*z$
4) $f(x,y,z) = -x + y*z - z$
5) $f(x,y,z) = -x*y + z$
6) $f(x,y,z) = -x - y + z*y$
7) $f(x,y,z) = x - y + z*z$
8) $f(x,y,z) = x*(-y) + z$
9) $f(x,y,z) = (-x)*y + z$
10) $f(x,y,z) = x*z - y + z$

Практическое занятие № 4. Двоично-десятичная арифметика

1 Цель работы

Исследовать с помощью отладчика работу арифметических команд. Научиться использовать команды двоично-десятичной арифметики.

2 Краткая теория

Арифметико-логическое устройство микропроцессора поддерживает две категории арифметических команд:

- команды двоичной целочисленной арифметики;
- команды двоично-десятичной арифметики.

2.1 Форматы двоично-десятичных целых чисел

Существует два формата представления двоично-десятичных (BCD – Binary Coded Decimal) целых чисел (рисунок 4.1):

- Упакованный двоично-десятичный формат целых чисел (packed BCD integers или просто BCD);
- Неупакованный двоично-десятичный формат целых чисел (unpacked BCD integers или ASCII).

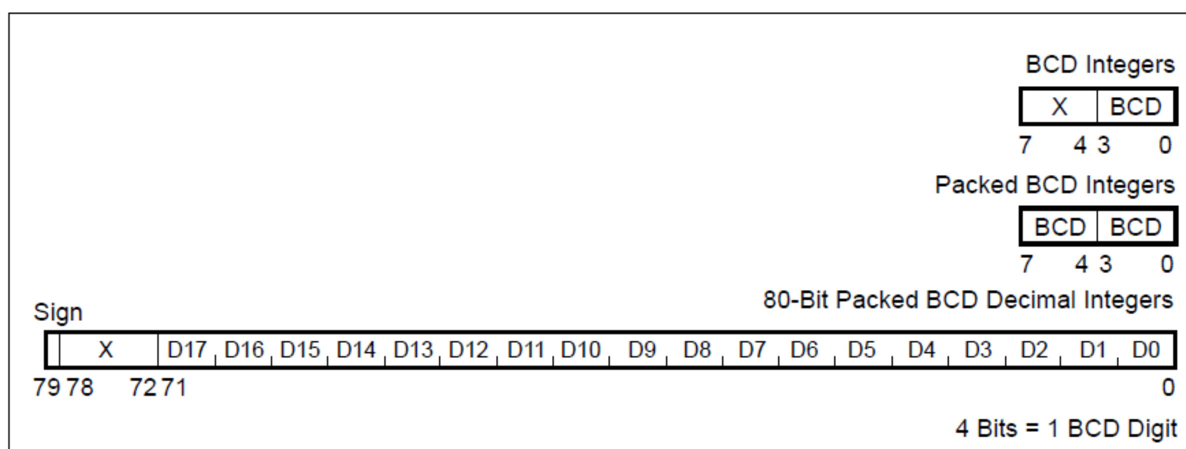


Рисунок 4.1 – Форматы двоично-десятичных целых чисел

2.2 Упакованный двоично-десятичный формат (BCD)

Упакованный двоично-десятичный формат чисел BCD представляет собой число, каждые четыре бита которого содержат только десятичные цифры от 0 до 9. В упакованных двоично-десятичных числах каждый байт числа содержит одну десятичную цифру.

Например, число 1648 в BCD-формате будет представлено в виде двух байт со значениями 0x16 и 0x48.

Длина BCD представления в два раза меньше ASCII-представления.

Обычные операции над такими числами приводят к неправильным результатам. Для коррекции результатов в состав микропроцессора входят

специальные команды двоично-десятичной арифметики. Обработка чисел осуществляется по одному байту.

Над числами в BCD-формате можно выполнять только операции сложения и вычитания, а для коррекции результата определены соответственно две команды:

1. Команда BCD-коррекции после сложения DAA
2. Команда BCD-коррекции после вычитания DAS

Обе команды не имеют операндов и выполняют коррекцию над результатом операции, располагаемым в регистре AL.

Пример 4.1 Работа с числами в BCD-формате

```
; BCD: 19 + 1 = 20
mov AL, 19h      ; AL = 19h
inc AL           ; AL = 1Ah
daa              ; AL = 20h

; BCD: 20 - 1 = 19
mov AL, 20h      ; AL = 20h
dec AL           ; AL = 1Fh
das              ; AL = 19h
```

Для BCD чисел занимающих больших одного байта обработка ведётся побайтно.

2.3 Неупакованный двоично-десятичный формат (ASCII)

Неупакованный двоично-десятичный формат чисел ASCII представляет собой число, младшие четыре бита которого содержат только десятичные цифры от 0 до 9, а старшие 4 бита не используются. В упакованных двоично-десятичных числах каждый байт числа содержит одну десятичную цифру.

Например, число 1648 в ASCII-формате будет представлено в виде четырех байт со значениями 0x01, 0x06, 0x04 и 0x08.

Обработка чисел в ASCII-формате также осуществляется по одному байту.

Над числами в ASCII-формате можно выполнять операции сложения, вычитания умножения и деления. Для коррекции результата определены четыре команды:

1. Команда ASCII-коррекции после сложения AAA
2. Команда ASCII-коррекции после вычитания AAS
3. Команда ASCII-коррекции после умножения AAM
4. Команда ASCII-коррекции перед делением AAD

Все эти команды не имеют операндов и выполняют коррекцию над результатом операции, располагаемым в регистре AL.

Пример 4.2 Работа с числами в ASCII-формате

```
; ASCII: 5 + 7 = 12
```

```
mov AX, 0005h   ; AH:AL = 05h
mov BL, 07h     ; BL = 07h
add AL, BL      ; AH:AL = 00:0Ch
aaa             ; AH:AL = 01:02h

; ASCII: 25 - 7 = 18
mov AX, 0205h   ; AH:AL = 02:05h
mov BL, 07h     ; BL = 07h
sub AL, BL      ; AH:AL = 01:FEh
aas             ; AH:AL = 01:08h

; ASCII: 5 * 7 = 35
mov AX, 0005h   ; AH:AL = 05h
mov BL, 07h     ; BL = 07h
mul BL          ; AH:AL = 00:23h
aam             ; AH:AL = 03:05h

; ASCII: 25 / 5 = 5
mov AX, 0205h   ; AH:AL = 02:05h
mov BL, 05h     ; BL = 05h
aad             ; AH:AL = 00:19h
div BL          ; AL = 05h
```

Для ASCII чисел занимающих больших одного байта обработка ведётся побайтно.

2.4 Преобразования форматов

На практике выполнение арифметических операций над числами в ASCII- и BCD-форматах не очень удобно. В большинстве случаев проще воспользоваться двоичным форматом, а затем выполнить преобразование в один из BCD-форматов.

2.4.1 Преобразование ASCII-формата в двоичный формат

Преобразование основывается на том, что ASCII-формат имеет основание 10, а компьютер выполняет арифметические операции только над числами с основанием 2. Процедура преобразования заключается в следующем: начиная с самого младшего байта числа в ASCII-формате, каждый байт последовательно умножают на соответствующую номеру байта степень 10-ки и складывают результаты.

Например, для числа 1648, преобразование будет следующим:

```
08 * 10**0 =    8 =   8h
04 * 10**1 =   40 =  28h
06 * 10**2 =  600 = 258h
01 * 10**3 = 1000 = 3E8h
             1648 = 670h
```

2.4.2 Преобразование двоичного формата в ASCII-формат

Наиболее часто ASCII-формат используется для отображения в консоли результатов вычислений. Это связанно с тем, что таблица символов консоли последовательно содержит символы от ‘0’ до ‘9’, коды которых отличаются ровно на 30h от значений цифр числа представленного в ASCII-формате. Т.е. символ ‘0’ имеет код 30h, а ‘1’ - 31h, и т.д. Это дает возможность быстрого преобразования ASCII-числа в строку, которую можно вывести на экран. Для этого просто нужно добавить к каждой цифре такого числа 30h и получить на выходе код требуемого символа.

Операция преобразования представляет собой процесс обратный предыдущему. Вместо умножения используется деление двоичного числа на 10 до тех пор, пока результат не будет меньше 1. Остатки, которые лежат в границах от 0 до 9, образуют число в ASCII-формате. В качестве примера рассмотрим обратное преобразование числа 670h:

```
670h / 0Ah = A4h | 8h
 A4h / 0Ah = 10h | 4h
 10h / 0Ah =  1h | 6h
  1h / 0Ah =  0h | 1h
```

Остатки вместе образуют последовательность цифр числа в ASCII-формате, получаемых справа налево, т.е. от младшего разряда к старшему.

2.4.3 Преобразование BCD-формата в двоичный формат

Преобразование из BCD-формата в двоичный аналогично преобразованию из ASCII-формата за исключением того, что умножения на 10 поочередно производятся то над младшими, то над старшими 4 битами числа.

2.4.4 Преобразование двоичного формата в BCD-формат

Преобразование из двоичного в BCD-формат аналогично преобразованию в ASCII-формата за исключением того, что полученные остатки попарно объединяются в производятся то над младшими, то над старшими 4 битами числа.

2.4.4 Быстрое преобразование ASCII-формата

Для чисел в пределах от 0 до 99 для преобразования можно воспользоваться командами коррекции умножения и деления.

Так команда ASCII-коррекции умножения выполняет преобразование из двоичного в ASCII-формат, а команда ASCII-коррекции деления выполняет преобразование из ASCII-формата обратно в двоичный.

Пример 4.3 Быстрое преобразование ASCII-формата

```
mov AX, 23h     ; AH:AL = 00:23h
```

```
aam             ; AH:AL = 03:05h (AH <= AL/10; AL <= AL%10 )

mov AX, 0205h   ; AH:AL = 02:05h
aad             ; AH:AL = 00:19h (19h = 10*AH + AL)
```

3 Контрольные вопросы

1) Опишите двоично-десятичные форматы представления целых чисел.
2) Опишите BCD-формат целых чисел.
3) Опишите ASCII-формат целых чисел.
4) Назовите допустимые операции с числами в BCD-формате.
5) Назовите допустимые операции с числами в ASCII-формат.
6) Какие команды коррекции существуют для чисел BCD-формата?
7) Какие команды коррекции существуют для чисел ASCII-формата?
8) Как выполняется преобразование BCD-формата в двоичный?
9) Как выполняется преобразование ASCII-формата в двоичный?
10) Как выполняется преобразование двоичного в BCD-формат?
11) Как выполняется преобразование двоичного в ASCII-формат?
12) Как выполнить быстрое преобразование ASCII-формата?

4 Задание

1) Выполнить задание из пункта 5 в соответствии со своим вариантом.
2) Оформить отчёт.

5 Варианты заданий

Выполните один вариант задания из пункта 5.2 предыдущей работы, выполняя операции над числами в ASCII-формате.

При выполнении задания размер операндов считать равным 2 байтам (2 десятичных разряда в ASCII-формате).

Номер варианта выбрать в соответствии с номером студента в списке группы N + 3.

Практическое занятие № 5. Команды логических операций

1 Цель работы

Исследовать с помощью отладчика работу логических команд и научиться их применять.

2 Краткая теория

Микропроцессор поддерживает набор базовых команд двоичной логики, однако от языков высокого уровня указанные команды применяются к аргументам побитово, т.е. отсутствует разбиение на чисто логические и побитовые операции. Поэтому для задания логических (с точки зрения ЯВУ) переменных необходимо чтобы все биты числа имели одинаковое значение.

2.1 Логические операции

Микропроцессоры семейства x86 поддерживают следующий набор логических операций:

1. Логическое отрицание (NOT, «!»):
Результат - ИСТИНА, если аргумент - ЛОЖЬ
2. Конъюнкция/логическое умножение (AND, «&»):
Результат - ИСТИНА, если оба аргумента – ИСТИНА
3. Дизъюнкция/логическое сложение (OR, «|»):
Результат - ИСТИНА, если хотя бы один аргумент – ИСТИНА
4. Исключающее ИЛИ (XOR, «^»):
Результат - ИСТИНА, если только один аргумент - ИСТИНА.

p	q	p & q	p | q	p ^ q	!p
False	False	False	False	False	True
True	False	False	True	True	False
False	True	False	True	True	True
True	True	True	True	False	False

Рисунок 5.1 – Таблица логических операций

2.2 Логические команды

Логические команды могут быть 8-ми, 16-ти и 32-х разрядными, т.е. оперировать, соответственно, байтами, словами и двойными словами.

2.2.1 Команда not

Команда выполняет инверсию битов указанного в команде операнда. Команда not не влияет на флаги.

Форматы команды:

```
not r/m8
not r/m16
not r/m32
```

Пример 5.1 Работа команды not

```
mov AL, 00001111b
not AL              ; al = 11110000b
```

2.2.2 Команда and

Команда выполняет побитовую конъюнкцию операндов.

Форматы команды:

```
and AL, imm8
and AX, imm16
and EAX, imm32
and r/m8, imm8
and r/m16, imm16
and r/m32, imm32
and r/m16, imm8
and r/m32, imm8
and r/m8, r8
and r/m16, r16
and r/m32, r32
and r8, r/m8
and r16, r/m16
and r32, r/m32
```

Команда может быть использована для сброса битов.

Пример 5.2 Работа команды and

```
mov AL, 11111111b
and AL, 00001111b   ; AL = 00001111b
```

2.2.3 Команда or

Команда выполняет побитовую дизъюнкцию операндов.

Форматы команды те же, что и у and.

Команда может быть использована для установки битов.

Пример 5.3 Работа команды or

```
mov     AL, 00001111b
or      AL, 11110000b; AL = 11111111b
```

2.2.4 Команда xor

Команда выполняет побитовую операцию исключающего или над операндами.

Форматы команды те же, что и у and.

Команда может быть использована для обнуления регистров, а также для инвертирования битов.

Пример 5.4 Работа команды xor

```
xor     AX, AX      ; AX = 0000000000000000b

mov     AL, 00001111b
xor     AL, 11111111b; AL = 11110000b
```

2.2.5 Команда test

Команда выполняет операцию AND, но не изменяет операнд-приемник. Воздействует на флаги SF, ZF и PF.

Форматы команды те же, что и у AND.

2.3 Пример вычисления логического выражения

Вычислить значение выражения z = x or not (y and not (x xor y)), где x = true, y =false. (Очевидно, что ответ - true)

```
.586
.model flat, stdcall
option casemap :none   ; case sensitive

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

    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

.data
x   dw 0FFFFh   ; true
y   dw 00000h   ; false

.code
main:
    ; z = x or not ( y and not (x xor y)),
    mov AX, x   ; x = true
    mov BX, y   ; y = false
    mov CX, AX  ; Копия x
    xor CX, BX  ; x xor y
    not CX      ; not (x xor y)
    and CX, BX  ; y and ...
    not CX      ; not ( y and ...)
    or CX, AX   ; X or not(...)

quit:
    mov eax, 0
    invoke ExitProcess, eax
end main
```

3 Контрольные вопросы

1) Опишите таблицу истинности для основных логических операций.
2) Как сбросить определённый бит числа?

3) Как установить определённый бит числа?
4) Как инвертировать определённый бит числа?
5) Как обнулить значение в некотором регистре?

4 Задание

1) Выполнить задание из пункта 5 в соответствии с вариантом.
2) Оформить отчёт.

5 Варианты заданий

Выполните 5 (пять) вариантов из нижеприведенного списка.

Номер варианта выбрать в соответствии с номером студента в списке группы N, N+3, N+5, N+9, N+13.

Вычислить значение выражения:

1. (not(X or Y)) xor (X and Y)
2. (not X) or (X xor (X and Y))
3. Y xor (X and (not(X and Y)))
4. X and (Y xor (not(X or Y)))
5. (not(X and Y)) or (X xor Y)
6. X or (not(X xor (X and Y)))
7. Y and (not(X or (Y xor X)))
8. (not(X xor Y)) or (Y and X)
9. (X or (not Y)) xor (X and Y)
10. (X xor Y) and (X or (not X))
11. (X and (not Y)) or (Y xor X)
12. (X and Y) xor (not(X) or Y)
13. (not(X and Y)) or (X xor Y)
14. Not(X) or not(Y) xor X
15. (not X) or (not Y) and (X xor Y)
16. X xor (not (X and Y)) or Y
17. Y and (X or Y) xor Y
18. (not(X and Y)) xor Y or X
19. (X xor Y) or (not(X and Y))
20. (not(X or Y)) xor (X and Y)
21. (Y xor X) and (X or not(X))
22. (not X) xor (not Y) or (X) and (Y)
23. (Y) and (not(X)) or (not(X xor Y))
24. X xor (not X) or (Y and X)
25. (Y) or (Y and X) xor (not X)
26. Y or not(Y) xor (X and Y)
27. X or (X xor Y) and not(Y)
28. Y and not(X) xor (X or Y)
29. (Y) and not(X xor (Y or X))
30. (not X) or (not(X xor (Y and X)))

* Считать значения x = true,a y = false.

Практическое занятие № 6. Команды передачи управления

1 Цель работы

Изучить работу команд управления, сравнения и условной пересылки. Научиться использовать эти команды для организации программ с нелинейным потоком выполнения.

2 Краткая теория

Команды передачи управления делятся на команды условных переходов и безусловных переходов.

Безусловные переходы подразделяются на собственно переходы (без возврата в точку перехода) и вызовы подпрограмм (с возвратом после завершения подпрограммы).

Помимо этого команды переходов различают по «дальности» передачи управления.

2.1 Команда арифметического сравнения

Для арифметического сравнения операндов в языке ассемблера используется команда cmp, выполняющая сравнение источника и преемника путем вычитания.

```
cmp DST, SRC
```

Команда обновляет флаги OF, SF, ZF, AF, CF, но не изменяет значения операнда-источника и операнда назначения. Последующие команды Jcc, CMOVcc SETcc могут проверять флаги.

2.2 Команда логического сравнения

Для логического сравнения операндов используется команда test (Проверить), выполняющая сравнение источника и преемника путем применения логического "and" для операндов.

```
test DST, SRC
```

Команда очищает флаги OF и CF, оставляя флаг AF неопределенным и обновляя значения флагов SF, ZF и PF. Флаги могут быть проверены командами условной передачи управления или командами установки значения байта по условию. Разница между командами TEST и AND заключается в том, что команда TEST не изменяет значение операнда назначения. Разница между командами TEST и BT заключается в том, что TEST может проверять значения множества битов за одну операцию, в то время как команда BT проверяет один бит.

2.3 Команды условного перехода, условной установки байта и условной пересылки

Группы условных команд микропроцессора выполняют те или иные действия, если состояния регистра EFLAGS удовлетворяет условиям, заданным в команде.

В таблице 6.1 представлены команды условного перехода, условной установки байта и условной пересылки с указанием условий срабатывания.

Таблица 6.1 – Виды условных команд микропроцессора

Команды условного перехода	**Команды условной установки байта**	**Команды условной пересылки**	**Условие**	**Семантика**
ja jnbe	seta setnbe	cmova cmovnbe	CF=0 и ZF=0	выше не ниже и не равно
jae jnb jnc	setae setnb setnc	cmovae cmovnb cmovnc	CF=0	выше или равно не ниже нет переноса
jb jnae jc	setb setnae setc	cmovb cmovnae cmovc	CF=1	ниже не выше и не равно перенос
jbe jna	setbe setna	cmovbe cmovna	CF=1 или ZF=1	ниже или равно не выше
je jz	sete setz	cmove cmovz	ZF=1	равно ноль
jg jnle	setg setnle	cmovg cmovnle	ZF=0 и SF=OF	больше не меньше и не равно
jge jnl	setge setnl	cmovge cmovnl	SF=OF	больше или равно не меньше
jl jnge	setl setnge	cmovl cmovnge	SF<>OF	меньше не больше и не равно
jle jng	setle setng	cmovle cmovng	ZF=1 или SF<>OF	меньше или равно не больше
jne jnz	setne setnz	cmovne cmovnz	ZF=0	не равно не ноль
jno	setno	cmovno	OF=0	нет переполнения
jo	seto	cmovo	OF=1	переполнение
jnp jpo	setnp setpo	cmovnp cmovpo	PF=0	нет четности нечетное
jp jpe	setp setpe	cmovp cmovpe	PF=1	четность четное
jns	setns	cmovns	SF=0	нет знака
js	sets	cmovs	SF=1	знак
jcxz	-	-	CX=0	CX=0
jecx	-	-	ECX=0	ECX=0

Команды условного перехода используются для организации нелинейного потока управления в программах, изменяющих поведение в зависимости от выполнения тех или иных условий.

Встретив команду условного перехода, процессор проверяет соответствующие флаги и, в случае выполнения условия, осуществляет переход наказанную после команды метку.

Команды условного перехода являются короткими (short) и могут передавать управление только на +127/-128 байт.

Работая с командами условной установки, процессор устанавливает значение операнда размером байт в 1 или 0, в зависимости от выполнения или невыполнения условий.

Аналогично действует процессор по командам условной пересылки, выполняя запрашиваемую пересылку данных или нет.

> *Примечание*. Указанные выше команды легко запоминаются, если знать принципы их формирования.
> В общем виде команды представляются как Jcc, CMOVcc и SETcc, где cc представляет собой первые буквы условий срабатывания. Названия условий срабатывания происходят от английских слов.
> n – not – отрицание.
> e – equals – равно
> z – zero – ноль
> a – above – выше (беззнаковое больше)
> b – below – ниже (беззнаковое меньше)
> g – greater – больше (знаковое)
> l – less – меньше (знаковое)
> s – sign – знак
> p – parity – четность
> c – carry – перенос
> o – overflow – переполнение
> Большая часть остальных значений получается путем комбинации указанных, хотя не все возможные варианты существуют.
> При этом используется следующий порядок: сначала идет отрицание (если оно необходимо), затем само условие, а потом дополнительное условие равенства (если необходимо).

Рассмотрим ряд примеров демонстрирующих работу команд условного перехода.

Пример 6.1 Вычисление функции определения знака числа

```
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
```

```
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

; Сегмент данных
.data
x   dw 3
z   dw ?

; Сегмент кода
.code
main:

; Вычислим значение функции z = -1, если x < 0, и z = 1, если x >= 0.
; 1-й способ:
    mov AX, x
    mov BX, 0
    cmp AX, BX
    jge set_one
    mov z, -1
    jmp end_calcs
set_one:
    mov z, 1
end_calcs:

;2-й способ:
;(необходим компилятор, поддерживающий P6)
;.686
mov AX, x
xor BX, BX
mov DX, 1
cmp AX, BX
cmovge CX, DX
neg DX
cmp AX, BX
cmovl CX, DX
mov z, CX

quit:
    mov eax, 0
    invoke ExitProcess, eax
end main
```

Пример 6.2 Вычисление модуля числа

```
.686
.model flat, stdcall
option casemap :none    ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
```

```
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

; Сегмент данных
.data
x   dw -3
y   dw -5
z   dw ?

; Сегмент кода
.code
main:

;1-й способ:
    mov AX, x
    xor BX, BX
    cmp AX, BX
    jg save_result
    neg AX
save_result:
    mov z, AX

;2-й способ:
    mov AX, y
    xor BX, BX
change_sign:
    neg AX
    js change_sign
    mov z, AX

quit:
    mov eax, 0
    invoke ExitProcess, eax
end main
```

2.4 Команды работы с битами

2.4.1 Команды тестирования (сканирования) битов

Эти команды сканируют слово или двойное слово в поисках установленного бита и заносят в регистр номер первого установленного бита (целое число, определяющее позицию найденного бита). Сканируемая строка BitBase может находиться как в регистре, так и в памяти. Если все слово равно нулю, т.е. в нем нет единичных битов, устанавливается флаг ZF.

Если единичный бит найден, флаг ZF очищается. Если единичных битов не найдено, значение регистра назначения не определено. Состояние флагов OF, SF, ZF, PF и CF не определено.

Сканирование битов вперед просматривает биты от младшего к старшему (от бита 0 до старшего бита).

```
bsf r, BitBase
```

Сканирование битов в обратном порядке просматривает биты от старшего к младшему (от самого старшего бита к биту 0).

```
bsr r, BitBase
```

2.4.2 Команды проверки и модификации битов

Эти команды проверяют заданный номером BitOffset бит в строке BitBase и изменяют его значение при необходимости.

В общем виде синтаксис команд выглядит следующим образом:

```
BTcc r/m16/32/64, r16/32/64/imm8
```

Сохранить указанный бит в CF.

```
bt BitBase, BitOffset
```

Сохранить указанный бит в CF и установить его в 1.

```
bts BitBase, BitOffset
```

Сохранить указанный бит в CF и сбросить его в 0.

```
btr BitBase, BitOffset
```

Сохранить указанный бит в CF и инвертировать его.

```
btc BitBase, BitOffset
```

2.5 Команды безусловного перехода

Безусловные переходы осуществляются с помощью команды jmp, которая может использоваться в 5 разновидностях. Переход может быть:

- прямым коротким (в пределах -128... + 127 байтов);
- прямым ближним (в пределах текущего сегмента команд):
- прямым дальним (в другой сегмент команд);
- косвенным ближним (в пределах текущего сегмента команд через ячейку с адресом перехода);
- косвенным дальним (в другой сегмент команд через ячейку с адресом перехода).

Рассмотрим последовательно структуру программ с переходами разного вида.

2.5.1 Прямой короткий (short) переход

Прямым называется переход, в команде которого в явной форме указывается метка, на которую нужно перейти. Разумеется, эта метка должна присутствовать в том же программном сегменте, при этом помеченная ею команда может находиться как до, так и после команды jmp. Достоинство команды короткого перехода заключается в том, что она занимает лишь 2 байт памяти: в первом байте записывается код операции (EBh), во втором – смещение к точке перехода. Расстояние до точки перехода отсчитывается от очередной команды, т.е. команды, следующей за командой jmp. Поскольку требуется обеспечить переход как вперед, так и назад, смещение рассматривается, как число со знаком и, следовательно, переход может быть осуществлен максимум на 127 байт вперед или 128 байт назад. Прямой короткий переход оформляется следующим образом:

```
.code
…
jmp short go ;Код EB dd

…

go:
…
code ends
```

В комментарии указан код команды; dd (от displacement, смещение) обозначает байт со смещением к точке перехода от команды, следующей за командой jmp. При выполнении команды прямого короткого перехода процессор прибавляет значение байта dd к младшему байту текущего значения указателя команд IP (который, всегда указывает на команду, следующую за выполняемой). В результате в IP оказывается адрес точки перехода, а предложения, находящиеся между командой jmp и точкой перехода, не выполняются.

> *Примечание*. Конструкция с прямым переходом вперед часто используется для того, чтобы обойти данные, которые по каким-то причинам желательно разместить в сегменте команд.

2.5.2 Прямой ближний (near) переход

Прямой ближний (near) (или внутрисегментный) переход отличается от предыдущего только тем, что под смещение к точке перехода отводится целое слово (16 бит).

```
code segment
…
```

```
jmp go ;Код E9 dddd
…
go:
…
code ends
```

> *Примечание.* В 64-битном режиме работы процессора используется не 16-битное, а 32-битное смещение.

Метка go может находиться в любом месте сегмента команд, как до, так и после команды jmp. В коде команды dddd обозначает слово с величиной относительного смещения к точке перехода от команды, следующей за командой jmp.

При выполнении команды прямого ближнего перехода процессор должен прибавить значение слова dddd к текущему значению указателя команд IP и сформировать тем самым адрес точки перехода.

2.5.3 Прямой дальний (far) переход

Прямой дальний (far) (или межсегментный) переход позволяет передать управление в любую точку любого сегмента. При этом предполагается, что программа включает несколько сегментов команд. Команда дальнего перехода включает, кроме кода операции EAh, еще и полный адрес точки перехода, т.е. сегментный адрес и смещение. Транслятору надо сообщить, что этот переход – дальний (по умолчанию команда jmp транслируется, как команда ближнего перехода). Это делается с помощью описателя far ptr, указываемого перед именем точки перехода.

```
code1 segment

assume CS: code1 ;Сообщим транслятору, что это сегмент команд
…
jmp far ptr go ;Код EA dddd ssssы
…
code1 ends


code2 segment

assume CS : code2 ;Сообщим транслятору, что это сегмент команд
…
go:
…
code2 ends
```

Метка go находится в другом сегменте команд этой двухсегментной программы. В коде команды ssss – сегментный адрес сегмента code2, а dddd – смещение точки перехода go в сегменте команд code2.

Все виды прямых переходов требуют указания в качестве точки перехода программной метки. С одной стороны, это весьма наглядно;

просматривая текст программы, можно сразу определить, куда осуществляется переход. С другой стороны, такой переход носит статический характер – его нельзя настраивать по ходу программы. Еще более серьезный недостаток прямых переходов заключается в том, что они не дают возможность перейти по известному абсолютному адресу, т.е. не позволяют обратиться ни к системным средствам, ни вообще к другим загруженным в память программам. Действительно, программы операционной системы не имеют никаких меток, так как метка – это атрибут исходного текста программы, а программы операционной системы транслировались не нами и присутствуют в компьютере только в виде выполнимых модулей. А вот адреса каких-то характерных точек системных программ определить можно. Для обращения по абсолютным адресам надо воспользоваться командами косвенных переходов, которые, как и прямые, могут быть ближними и дальними.

2.5.4 Косвенный ближний переход

В отличие от команд прямых переходов, команды косвенных переходов могут использовать различные способы адресации и, соответственно, иметь много разных вариантов. Общим для них является то, что адрес перехода не указывается явным образом в виде метки, а содержится либо в ячейке памяти, либо в одном из регистров. Это позволяет при необходимости модифицировать адрес перехода, а также осуществлять переход по известному абсолютному адресу. Рассмотрим случай, когда адрес перехода хранится в ячейке сегмента данных. Если переход ближний, то ячейка с адресом состоит из одного слова и содержит только смещение к точке перехода.

```
; Сегмент данных
.data
jump_address    dd go ;Адрес перехода

; Сегмент кода
.code
main:

    jmp DS:jump_address     ;Код FF 26 dddd

...

    go: ;Точка перехода

...

quit:
```

Точка перехода go может находиться в любом месте сегмента команд. В коде команды dddd обозначает относительный адрес слова jump_address в сегменте данных, содержащем эту ячейку.

В приведенном фрагменте адрес точки перехода в слове jump_address задан однозначно указанием имени метки go. Такой вариант косвенного перехода выполняет фактически те же функции, что и прямой (переход по единственному заданному заранее адресу), только несколько более запутанным образом. Достоинства косвенного перехода будут более наглядны, если представить, что ячейка jump_address поначалу пуста, а по ходу выполнения программы внес, в зависимости от каких-либо условий, помещается адрес той или иной точки перехода:

```
    mov jump_address, offset go1
...
    mov jump_address, offset go2
...
    mov jump_address, offset go3
```

Разумеется, приведенные выше команды должны выполняться не друг за другом, а альтернативно. В этом случае создается возможность перед выполнением перехода определить или даже вычислить адрес перехода, требуемый в данных условиях.

Ассемблер допускает различные формы описания косвенного перехода через ячейку сегмента данных:

```
jmp DS:jump_address
jmp dword ptr jump_address
jmp jump_address
```

В первом варианте, использованном в приведенном выше фрагменте, указано, через какой сегментный регистр надлежит обращаться к ячейке jump_address, содержащей адрес перехода. Здесь допустима замена сегмента, если сегмент с ячейкой jump_address адресуется через другой сегментный регистр, например, ES или CS.

Во втором варианте подчеркивается, что переход осуществляется через ячейку размером в одно слово и, следовательно, является ближним. Ячейка jump_address объявлена с помощью директивы dd и содержит двухсловный адрес перехода, требуемый для реализации перехода. Описатель dword ptr перед именем ячейки с адресом перехода заставляет транслятор считать, что она имеет размер 2 слова (независимо от того, как она была объявлена).

Наконец, возможен и самый простой, третий вариант, который совпадает по форме с прямым переходом, но, тем не менее, является косвенным, так как символическое обозначение jump_address является именем поля данных, а не программной меткой. В этом варианте предполагается, что сегмент, в котором находится ячейка jump_address,

адресуется по умолчанию через регистр DS, хотя, как и во всех таких случаях, допустима замена сегмента. Тип перехода (ближний или дальний) определяется, исходя из размера ячейки jump_address. Однако этот вариант не всегда возможен. Для его правильной трансляции необходимо, чтобы транслятору к моменту обработки предложения

```
jmp jump_address
```

было уже известно, что собой представляет имя jump_address. Этого можно добиться двумя способами. Первый - расположить сегмент данных до сегмента команд, а не после, как в приведенном выше примере. Второй - заставить транслятор обрабатывать исходный текст программы не один раз, как он это делает по умолчанию, а несколько.

В приведенных примерах адрес поля памяти с адресом точки перехода задавался непосредственно в коде команды косвенного перехода. Однако этот адрес можно задать и в одном из регистров общего назначения (EBX, ESI или EDI). Для приведенного выше примера косвенного перехода в точку go, адрес которой находится в ячейке jump_address в сегменте данных, переход с использованием косвенной регистровой адресации может выглядеть следующим образом:

```
mov EBX, offset jump_address ;В EBX смещение поля с адресом перехода
jmp dword ptr [EBX] ;Переход в точку go
```

Особенно большие возможности предоставляет методика косвенного перехода с использованием базово-индексной адресации через пары регистров, например, [EBX][ESI] или [EBX][EDI]. Этот способ удобен в тех случаях, когда имеется ряд альтернативных точек перехода, выбор которых зависит от некоторых условий. В этом случае в сегменте данных создается не одно поле с адресом, а таблица адресов переходов. В базовый регистр EBX загружается адрес этой таблицы, а в один из индексных регистров – определенный тем или иным способом индекс в этой таблице. Переход осуществляется в точку, соответствующую заданному индексу. Структура программы, использующей такую методику, выглядит следующим образом:

```
; Сегмент данных
.data
jmp_tbl  label word ;Таблица адресов переходов
go1_addr dd go1 ;Адрес первой альтернативной точки перехода
go2_addr dd go2 ;Адрес второй альтернативной точки перехода
go3_addr dd go3 ;Адрес третьей альтернативной точки перехода

; Сегмент кода
.code
main:

    lea EBX, jmp_tbl ;Загружаем в EBX базовый адрес таблицы
    mov ESI, 4       ;Вычисленное каким-то образом смещение в таблице
```

```
    jmp dword ptr [EBX][ESI] ;Если индекс = 4, переход в точку go2
go1: ;1-я точка перехода
...
go2: ;2-я точка перехода
...
go3: ;3-я точка перехода
...
```

Приведенный пример носит условный характер; в реальной программе индекс, помещаемый в регистр ESI, должен вычисляться по результатам анализа некоторых условий.

Наконец, существует еще одна разновидность косвенного перехода, в котором не используется сегмент данных, а адрес перехода помещается непосредственно в один из регистров общего назначения. Часто такой переход относят к категории прямых, а не косвенных, однако это вопрос не столько принципа, сколько терминологии.

Применительно к обозначениям последнего примера такой переход будет выглядеть, например, следующим образом:

```
mov EBX, offset go1
jmp EBX
```

Здесь, как и в предыдущих вариантах, имеется возможность вычисления адреса перехода, однако нельзя этот адрес индексировать.

2.5.5 Косвенный дальний (межсегментный) переход

Как и в случае ближнего перехода, переход осуществляется по адресу, который содержится в ячейке памяти, однако эта ячейка содержит полный (сегмент плюс смещение) адрес точки перехода. Программа в этом случае должна включать, по меньшей мере, два сегмента команд.

Как и в случае ближнего косвенного перехода, ассемблер допускает различные формы описания дальнего косвенного перехода через ячейку сегмента данных:

```
jmp DS:jmp_addr ;Возможна замена сегмента
jmp fword ptr jmp_addr ;Если поле jmp_addr объявлено операторами df
jmp jmp_addr ;Характеристики ячейки должны быть известны
```

Для дальнего косвенного перехода, как и для ближнего, допустима адресация через регистр общего назначения, если в него поместить адрес поля с адресом перехода:

```
mov EBX,offset jmp_addr
jmp fword ptr [EBX]
```

Возможно также использование базово-индексной адресации, если в сегменте данных имеется таблица с двухсловными адресами точек переходов.

2.5.6 Явление оборачивания адреса

Рассматривая вычисление адреса точки перехода, следует иметь в виду явление оборачивания, суть которого можно кратко выразить такими соотношениями:

```
FFFFFFFFh + 00000001h = 00000000h
00000000h - 00000001h = FFFFFFFFh
```

Если последовательно увеличивать содержимое какого-либо регистра или ячейки памяти, то, достигнув верхнего возможного предела 0xFFFFFFFF, число "перевалит" через эту границу, станет равным нулю и продолжит нарастать в области малых положительных чисел (1, 2, 3, и т.д.). Точно так же, если последовательно уменьшать некоторое положительное число, то оно, достигнув нуля, перейдет в область отрицательных (или, что то же самое, больших беззнаковых) чисел, проходя значения 2, 1, 0, FFFFFFFFh, FFFFFFFEh и т.д.

Таким образом, при вычислении адреса точки перехода смещение следует считать числом без знака, но при этом учитывать явление оборачивания. Если команда jmp находится где-то в начале сегмента команд, а смещение имеет величину порядка 2**32, то переход произойдет вперед, к концу сегмента. Если же команда находится в конце сегмента команд, а смещение имеет ту же величину порядка 2**32, то для определения точки перехода надо двигаться по сегменту вперед, дойти до его конца и продолжать перемещаться от начала сегмента по-прежнему вперед, пока не будет пройдено заданное в смещении число байтов. Для указанных условий мы попадем в точку, находящуюся недалеко от команды jmp со стороны меньших адресов.

2.6 Циклы

Команды управления циклом являются командами условного перехода, которые используют значение, помещенное в регистр ECX, в качестве счетчика числа выполнений тела цикла. Все команды управления циклом уменьшают значение регистра ECX при каждом выполнении цикла и завершают работу при достижении значения ноль. Четыре из пяти команд управления циклом воспринимают флаг ZF в качестве условия завершения цикла до момента достижения счетчиком цикла значения ноль.

LOOP label (Цикл до тех пор, пока ECX не равен нулю) является командой условной передачи управления, которая уменьшает содержимое регистра ECX перед проверкой условия завершения цикла. Если содержимое регистра ECX отлично от нуля, программа передает управление

по адресу, указанному в команде в качестве адреса назначения (преемника). Команда LOOP приводит к выполнению части программы, которое повторяется до тех пор, пока счетчик не станет равным нулю. Когда достигается значение нуля, выполнение передается команде, следующей непосредственно за командой LOOP.

Если значение регистра ECX равняется нулю перед первым выполнением цикла, счетчик цикла уменьшается на 1, регистру присваивается значение 0FFFFFFFFH и цикл выполняется 2**32 раза.

LOOPE lable (Цикл до тех пор, пока равенство) и LOOPZ (Цикл до тех пор, пока ноль) являются синонимами одной и той же команды. Эти команды являются командами условного перехода, которые уменьшают содержимое регистра ECX перед проверкой условия завершения цикла. Если значение регистра ECX не равно нулю и установлен флаг ZF, программа передает управление по адресу, указанному в качестве операнда назначения в команде. Когда достигнуто значение нуля или флаг ZF очищен, выполнение передается команде, следующей непосредственно за командой LOOPE/LOOPZ.

LOOPNE label (Цикл до тех пор, пока неравенство) и LOOPNZ (Цикл до тех пор, пока не ноль) являются синонимами одной и той же команды. Эти команды являются командами условного перехода, которые уменьшают содержимое регистра ECX перед проверкой условия завершения цикла. Если значение регистра ECX не равно нулю и очищен флаг ZF, программа передает управление по адресу, указанному в качестве операнда назначения в команде. Когда достигнуто значение нуля или флаг ZF установлен, выполнение передается команде, следующей непосредственно за командой LOOPNE/LOOPNZ.

Для организации циклического выполнения участка кода используется команда loop. Рассмотрим технику использования этой команды на следующем примере.

Пример 6.3 Вычисление суммы чисел от 1 до n

```
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc

    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

; Сегмент данных
.data
n   dd  10
; Сегмент кода
```

```
.code
main:
    ; вычислить сумму целых чисел от 1 до N.
    xor EAX, EAX
    mov ECX, n ; количество повторов

sum_values:
    add EAX, ECX
    loop sum_values ; пока ECX не равен нулю

quit:
    mov eax, 0
    invoke ExitProcess, eax
end main
```

3 Контрольные вопросы

1) Опишите работу команды арифметического сравнения?
2) Опишите работу команды логического сравнения?
3) Опишите работу команд работы с битами?
4) Какие виды команд передачи управления вы знаете?
5) Назовите известные вам команды условного перехода?
6) Назовите известные вам команды условной пересылки?
7) Назовите известные вам команды условной установки байта?
8) Какие виды безусловных переходов вы знаете?
9) Чем прямой переход отличается от косвенного?
10) В чем отличие дальнего перехода от ближнего и короткого?
11) В чем отличие ближнего перехода от короткого?
12) Какие команды предназначены для организации циклов?
13) Опишите алгоритм работы команды loop?
14) Что представляет собой явления циклического оборачивания адреса?

4 Задание

1) Выполнить задание в соответствии с пунктом 5.
2) Оформить отчёт.

5 Задание для выполнения работы

1) Вычислить и вывести на экран значения функции F.

$$F = \begin{cases} ax^2 + b, при \quad x < 0 \quad и \quad b \neq 0 \\ \dfrac{x-a}{x-c}, при \quad x > 0 \quad и \quad b = 0 \\ \dfrac{x}{c}, в \quad остальных \quad случаях \end{cases}$$

2) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases}\dfrac{1}{ax}-b, при\ \ x+5<0\ \ и\ \ c=0\\ \dfrac{x-a}{x}, при\ \ x+5>0\ \ и\ \ c\neq 0\\ \dfrac{10x}{c-4}, в\ \ остальных\ \ случаях\end{cases}$$

3) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases}-ax-c, при\ \ c<0\ \ и\ \ x\neq 0\\ \dfrac{x-a}{-c}, при\ \ c>0\ \ и\ \ x=0\\ \dfrac{bx}{c-a}, в\ \ остальных\ \ случаях\end{cases}$$

4) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases}a-\dfrac{x}{10+b}, при\ \ x<0\ \ и\ \ b\neq 0\\ \dfrac{x-a}{x-c}, при\ \ x>0\ \ и\ \ b=0\\ 3x+\dfrac{2}{c}, в\ \ остальных\ \ случаях\end{cases}$$

5) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases}ax^2+b^2x, при\ \ c<0\ \ и\ \ b\neq 0\\ \dfrac{x+a}{x+c}, при\ \ c>0\ \ и\ \ b=0\\ \dfrac{x}{c}, в\ \ остальных\ \ случаях\end{cases}$$

6) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases}-ax^2-b, при\ \ x<5\ \ и\ \ c\neq 0\\ \dfrac{x-a}{x}, при\ \ x>5\ \ и\ \ c=0\\ \dfrac{-x}{c}, в\ \ остальных\ \ случаях\end{cases}$$

7) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} -ax^2, при \quad c<0 \quad и \quad a\neq 0 \\ \dfrac{a-x}{cx}, при \quad c>0 \quad и \quad a=0 \\ 1+\dfrac{x}{c}, в \quad остальных \quad случаях \end{cases}$$

8) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^2+b^2x, при \quad a<0 \quad и \quad x\neq 0 \\ x-\dfrac{a}{x-c}, при \quad a>0 \quad и \quad x=0 \\ 1+\dfrac{x}{c}, в \quad остальных \quad случаях \end{cases}$$

10) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^2-bx+c, при \quad x<3 \quad и \quad b\neq 0 \\ \dfrac{x-a}{x-c}, при \quad x>3 \quad и \quad b=0 \\ \dfrac{x}{c}, в \quad остальных \quad случаях \end{cases}$$

11) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^2+\dfrac{b}{c}, при \quad x<1 \quad и \quad c\neq 0 \\ \dfrac{x-a}{(x-c)^2}, при \quad x>1{,}5 \quad и \quad c=0 \\ \dfrac{x^2}{c^2}, в \quad остальных \quad случаях \end{cases}$$

12) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^3+b^2+c, при \quad x<0{,}6 \quad и \quad b+c\neq 0 \\ \dfrac{x-a}{x-c}, при \quad x>0{,}6 \quad и \quad b+c=0 \\ \dfrac{x}{c}+\dfrac{x}{a}, в \quad остальных \quad случаях \end{cases}$$

13) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^2+b, при \;\; x-1<0 \;\; и \;\; b-x\neq 0 \\ \dfrac{x-a}{x}, при \;\; x-1>0 \;\; и \;\; b+x=0 \\ \dfrac{x}{c}, в \;\; остальных \;\; случаях \end{cases}$$

14) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} -ax^2+b, при \;\; x<0 \;\; и \;\; b\neq 0 \\ \dfrac{x}{x-c}+5{,}5, при \;\; x>0 \;\; и \;\; b=0 \\ \dfrac{x}{-c}, в \;\; остальных \;\; случаях \end{cases}$$

15) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} a(x+c)^2-b, при \;\; x=0 \;\; и \;\; b\neq 0 \\ \dfrac{x-a}{-c}, при \;\; x=0 \;\; и \;\; b=0 \\ a+\dfrac{x}{c}, в \;\; остальных \;\; случаях \end{cases}$$

16) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^2-cx+b, при \;\; x+10<0 \;\; и \;\; b\neq 0 \\ \dfrac{x-a}{x-c}, при \;\; x+10>0 \;\; и \;\; b=0 \\ \dfrac{-x}{a-c}, в \;\; остальных \;\; случаях \end{cases}$$

17) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} ax^3+bx^2, при \;\; x<0 \;\; и \;\; b\neq 0 \\ \dfrac{x-a}{x-c}, при \;\; x>0 \;\; и \;\; b=0 \\ \dfrac{x+5}{c(x-10)}, в \;\; остальных \;\; случаях \end{cases}$$

18) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} a(x+7)^2-b, при \quad x<5 \quad и \quad b\neq 0 \\ \dfrac{x-cd}{ax}, при \quad x>5 \quad и \quad b=0 \\ \dfrac{x}{c}, в \quad остальных \quad случаях \end{cases}$$

19) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} -\dfrac{2x-c}{cx-a}, при \quad x<0 \quad и \quad b\neq 0 \\ \dfrac{x-a}{x-c}, при \quad x>0 \quad и \quad b=0 \\ -\dfrac{x}{c}+\dfrac{-c}{2x}, в \quad остальных \quad случаях \end{cases}$$

20) Вычислить и вывести на экран значения функции F.

$$F=\begin{cases} -\sqrt[3]{ax}-\dfrac{c^2}{b}, при \quad c<0 \quad и \quad x\neq 0 \\ \dfrac{x-a}{-c}+\ln^2 x, при \quad c>0 \quad и \quad x=0 \\ \dfrac{bx}{c-a}+8^x, в \quad остальных \quad случаях \end{cases}$$

Практическое занятие № 7. Команды сдвигов

1 Цель работы

Изучить работу команд сдвигов микропроцессора и научиться их применять.

2 Краткая теория

Команды сдвига и циклического сдвига переставляют биты внутри операнда. Эти команды подразделяются на 3 группы:

- Команды линейного сдвига.
- Команды сдвига с двойной точностью.
- Команды циклического сдвига.

По направлению сдвига команды делятся на команды сдвига влево и, соответственно, вправо.

Команды сдвига могут быть 8-ми, 16-ти и 32-х разрядными.

В зависимости от разрядности могут сдвигать, соответственно, на 7, 15 или 31 разряд.

Количество сдвигаемых бит может указываться непосредственным значением или в регистре CL. При этом флаг CF содержит значение последнего выдвинутого бита.

2.1 Команды линейного сдвига

Команды линейного сдвига делятся на команды арифметического и команды логического сдвига.

Арифметический сдвиг вправо копирует знаковый бит в пустую позицию старшего бита операнда, в то время как логический сдвиг вправо, сдвинув операнд вправо, очищает пустые позиции. Арифметический сдвиг является самым быстрым способом выполнения простейших вычислений. Например, арифметический сдвиг вправо на один бит выполняет деление целого на два. Логический сдвиг делит целое без знака или положительное целое, но отрицательное целое со знаком теряет свой знаковый бит.

Команды арифметического и логического сдвига вправо, SAR и SHR, отличаются друг от друга только своей интерпретацией позиций битов, освобождаемых при смещении содержимого операндов. При этом нет различий в командах логического и арифметического сдвига влево и два символьных имени, SAL и SHL, поддерживаются языком ассемблера для обозначения одной команды.

```
SH/AR/L r/m8/16/32, imm8/cl/1
```

Счетчик указывает число битовых позиций, на которое надо сдвинуть операнд. Биты могут быть сдвинуты максимум на 31 позицию. Команды сдвига могут задавать счетчик сдвига любым из трех способов. Одна форма команд сдвига всегда выполняет сдвиг на один бит. Вторая форма задает

счетчик сдвига как непосредственное значение. Третья форма задает счетчик как значение, содержащееся в регистре CL. Последняя форма позволяет задавать счетчик как результат вычислений. Используются только пять младших битов (разрядов) регистра CL.

Флаг CF заполняется значением последнего бита, вытесненного за границы операнда. В командах сдвига на один бит флаг OF устанавливается равным единице, если значение самого старшего бита (знакового бита) изменяется в процессе операции. В противном случае флаг OF очищается (присваивается значение ноль). После сдвига более чем на одну позицию значение флага OF не определено. При сдвиге на одну или более позиций изменяются значения флагов SF, ZF, PF и CF, и состояние флага AF не определено.

2.1.1 Команды логического сдвига

Команды SHR/SHL логического сдвига вправо/влево осуществляют сдвиг битов с потерей выдвинутых битов и обнулением освободившихся.

Форматы команд:

```
shr     r/m8, 1
shr     r/m8, CL
shr     r/m8, imm8
shr     r/m16, 1
shr     r/m16, CL
shr     r/m16, imm8
shr     r/m32, 1
shr     r/m32, CL
shr     r/m32, imm8
```

Могут применяться для умножения/деления беззнаковых целых чисел на степени числа 2 (2d=10b).

Пример 7.1 Быстрое умножение на степень 2

```
; Умножение содержимого регистра AX на 8.
shl     AX, 3
```

Считается, что применение команд сдвига вместо команд умножения/деления эффективнее с точки зрения быстродействия.

```
; Кроме того, две команды
shl     AX, 1
shl     AX, 1
; эффективнее команд
mov     CL, 2
shl     AX, CL
```

2.1.2 Команды арифметического сдвига

Команды SAR/SAL арифметического сдвига вправо/влево учитывают в своей работе знаковый разряд. При сдвиге вправо оставляют на месте и размножают знаковый разряд.

Результат сдвига вправо не совпадает с результатом деления, выполняемого командой idiv.

Форматы команд те же, что и у SHR/SHL.

2.2 Команды сдвига с двойной точностью

Эти команды обеспечивают основные операции, необходимые для выполнения действий над длинными невыровненными битовыми строками.

Приемник сдвигается вправо/влево на число бит указанное в счетчике. Старший (для SHRD) или младший (для SHLD) бит не обнуляется, а считывается из источника, значение которого не изменяется.

Из двух операндов, операнд-источник должен быть регистром, в то время как операнд назначения может быть как регистром, так и ячейкой памяти. Количество битов, на которое выполняется сдвиг, может быть задано в регистре CL или непосредственно значением байта в команде. Биты, вытесненные за границу операнда-источника, заполняют пустые биты в операнде назначения, который тоже сдвигается. Сохраняется значение только операнда-преемника.

Когда выполняется сдвиг на ноль позиций, ни один из флагов не подвергается изменениям. В противном случае, флагу CF присваивается значение последнего бита, вытесненного из операнда назначения, и изменяются значения флагов SF, ZF и PF. При сдвиге на один разряд флаг OF получает значение "единица", если знак операнда изменяется, в противном случае флаг OF очищается. При сдвигах более чем на один бит состояние флага AF неопределено.

```
shld DST, SRC, QTY
```

Двойной сдвиг влево сдвигает биты операнда назначения влево, заполняя пустые биты значениями битов, вытесняемых из операнда-источника. Результат запоминается в операнде назначения. Операнд-источник не изменяется.

```
shrd DST, SRC, QTY
```

Двойной сдвиг вправо сдвигает биты операнда назначения вправо, заполняя пустые биты значениями битов, вытесняемых из операнда-источника. Результат запоминается в операнде назначения. Операнд-источник не изменяется.

Форматы команд

```
shrd r/m16, r16, imm8
```

```
shrd r/m32, r32, imm8
shrd r/m16, r16, CL
shrd r/m32, r32, CL
```

Рассмотрим пример работы команды.

```
; Например, если приемник содержит 00101001b,
; источник - 00001010b, счетчик - 3,
; SHRD даст в результате 01000101b,
; а SHLD - 01001000b.

;   DST    |    SRC        =>   DST    |    SRC
; 00101001|00001010 shrd => 01000101|00001010
; 00101001|00001010 slrd => 01001000|00001010
```

2.3 Команды циклического сдвига

Различают две команды циклического сдвига:

- обычный циклический сдвиг;
- циклический сдвиг через перенос.

Команды циклического сдвига выполняют циклическое перемещение разрядов в байтах, словах и двойных словах. Биты, вытесненные с одного конца операнда, заносятся в него с другого конца. В отличие от команд смещения никакие биты не очищаются в процессе циклического сдвига.

Команды циклического сдвига используют только флаги CF и OF. Флаг CF может работать в качестве расширения операнда в двух командах циклического сдвига, позволяющих биту стать обособленным и затем быть проверенным командами условного перехода (JC или JNC). Флаг CF всегда содержит значение последнего бита, вытесненного за пределы операнда в процессе циклического сдвига, даже если команда не использует флаг CF в качестве расширения операнда. Состояние флагов SF, ZF, AF и PF не изменяется.

При циклическом сдвиге на один бит флаг OF устанавливается, если операция изменяет самый старший бит (знаковый бит) операнда назначения. Если сохраняется исходное значение знака, флаг OF очищается. После циклического сдвига более чем на один бит значение флага OF не определено.

2.3.1 Команды циклического сдвига

Команды циклического сдвига ROL/ROR осуществляют сдвиг битов без потери выдвинутых битов. Выдвинутые биты переносятся на место освободившихся.

```
ROL DST, QTY
```

Циклический сдвиг влево циклически сдвигает байт, слово или двойное слово операнда назначения влево на один бит или на количество битов, заданное в операнде-счетчике (непосредственное значение или

значение, содержащееся в регистре CL). Для каждого разряда бит, который вытесняется с левого конца операнда, возвращается в правый конец.

```
ROR DST, QTY
```

Циклический сдвиг вправо циклически сдвигает байт, слово или двойное слово операнда назначения вправо на один бит или на количество битов, заданное в операнде-счетчике (непосредственное значение или значение, содержащееся в регистре CL). Для каждого разряда бит, который вытесняется с правого конца операнда, возвращается в левый конец.

Форматы команд

```
ror r/m8, 1
ror r/m8, CL
ror r/m8, imm8
ror r/m16, 1
ror r/m16, CL
ror r/m16, imm8
ror r/m32, 1
ror r/m32, CL
ror r/m32, imm8
```

Команды циклического сдвига можно применять для обмена значениями младшей и старшей половины байта.

```
mov AL, 11110000b
ror AL, 4; AL = 00001111b
```

2.3.2 Команда циклического сдвига через перенос

Команда RCR/RCL циклического сдвига вправо/влево отличаются от команд ROR/ROL тем, что флаг CF рассматривают как расширение приемника. Выдвинутый бит помещается в CF, а на следующем такте пересылается в самый младший бит.

```
RCL DST, QTY
```

Циклический сдвиг влево через перенос циклически сдвигает байт, слово или двойное слово операнда назначения влево на один бит или на количество битов, заданное в операнде-счетчике (непосредственное значение или значение, содержащееся в регистре CL).

Эта команда отличается от ROL тем, что она интерпретирует флаг CF как однобитовое расширение операнда-преемника со стороны старших разрядов. Для каждого разряда бит, который вытесняется с левого конца операнда, перемещается во флаг CF. В тоже время бит, содержащийся во флаге CF, вводится с правой стороны.

```
RCR DST, QTY
```

Циклический сдвиг вправо через перенос циклически сдвигает байт, слово или двойное слово операнда назначения вправо на один бит или на количество битов, заданное в операнде-счетчике (непосредственное значение или значение, содержащееся в регистре CL).

Эта команда отличается от ROR тем, что она интерпретирует флаг CF как однобитовое расширение операнда-преемника со стороны младших разрядов. Для каждого разряда бит, который вытесняется с правого конца операнда, перемещается во флаг CF. В тоже время бит, содержащийся во флаге CF, вводится с левой стороны.

3 Контрольные вопросы

1) Какие разновидности команд сдвига вы знаете?
2) Назовите команды линейного сдвига.
3) Назовите команды циклического сдвига.
4) Опишите работу команды сдвига с двойной точностью.
5) В чем отличие команд ROR/ROL от RCR/RCL?
6) В чем отличие команд SAR от SHR?
7) Как быстро умножить число на степень двойки?
8) Как обменять значениями младшую и старшую часть байта/слова/двойного слова?

4 Задание

1) Выполнить задание в соответствии с пунктом 5.
2) Оформить отчёт.

5 Задание для выполнения работы

1) Подсчитать количество установленных бит в числе.
2) Подсчитать количество сброшенных бит в числе.
3) Подсчитать количество установленных бит в четных позициях числа.
4) Подсчитать количество установленных бит в нечетных позициях числа.
5) Подсчитать количество сброшенных бит в четных позициях числа.
6) Подсчитать количество сброшенных бит в нечетных позициях числа.

Практическое занятие № 8. Подпрограммы и вызов процедур

1 Цель работы

Изучить принципы создания подпрограмм на языке ассемблера и исследовать работу команд операций с процедурами.

2 Краткая теория

На практике программирования часто встречается ситуация, когда одну и ту же группу операторов, реализующих определенный алгоритм, требуется повторить без изменений в нескольких других местах программы. Такую группу операторов во всех языках программирования принято называть подпрограммой, ее можно вызвать для исполнения по имени любое количество раз из различных мест программы.

2.1 Создание подпрограмм на языке ассемблера

Для объявления подпрограммы на языке ассемблера используется пара ключевых слов proc и endp. Место объявления подпрограммы (процедуры в терминах языка) должно располагаться вне основной части программы (в ассемблере не допускается вложенность процедур).

Процедура оформляется следующим образом:

```
метка   proc    тип_адресации
        ;тело процедуры
метка   endp
```

Имя процедуры представляет собой метку части кода, которой будет передаваться управление при вызове. Поэтому объявление процедуры в таком виде эквивалентно объявлению обычной метки, расположенной за пределами основного потока управления.

```
метка:
        ;тело процедуры
```

Основную часть программы также можно оформлять в виде процедуры.

Пример 8.1 Оформление основной части в виде процедуры

```
main    proc far
        ; тело основной программы
main    endp

end     main
```

При этом команда ret не используется, а в качестве точки входа необходимо указать имя главной процедуры.

2.2 Вызов процедур

Команда вызова процедуры call передает управление и сохраняет адрес команды, следующей за командой call, для дальнейшего использования командой ret (Возврат). call сохраняет текущее содержимое регистра EIP в стеке. Команда ret в вызванной процедуре использует этот адрес в стеке для передачи управления назад в вызывающую программу.

```
call target
```

Различают ближний (near) дальний (far) варианты команды call.

Помимо этого команд call позволяет задавать абсолютный и относительный адрес передачи управления.

Косвенная команда call указывает абсолютный адрес одним из следующих способов:

1. Программа может выполнить переход по адресу в регистре общего назначения. 32-разрядное значение копируется в регистр EIP, адрес возврата сохраняется в стеке и выполнение продолжается.

2. Адресат назначения может быть переменной в памяти, указанной с использованием стандартного режима адресации. Операнд копируется в регистр EIP, адрес возврата сохраняется в стеке и выполнение продолжается.

2.3 Команда возврата

Возврат из процедуры завершает выполнение процедуры и передает управление команде, следующей за командой call, которая вызвала данную процедуру. Команда ret восстанавливает содержимое регистра EIP, которое было сохранено в стеке при вызове процедуры.

```
RET [num]
```

Команда ret имеет необязательный операнд непосредственного значения. Когда операнд имеется, эта константа прибавляется к содержимому регистра ESP, что имеет эффект удаления всех параметров, сохраненных в стеке перед вызовом процедуры.

2.4 Пример работы с процедурами

Пример 8.2 Объявление и вызов процедуры

```
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc
```

```
    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib

; Сегмент данных
.data
n   dd  10

; Сегмент кода
.code
sum_n proc
    ; Вычисляет сумму целых чисел от 1 до N.
    ; В регистре ECX- число значений для суммирвоания
    ; Результат в EAX

    xor EAX, EAX

sum_values:
    add EAX, ECX
    loop sum_values ; пока ECX не равен нулю

    ret
sum_n endp

main proc far

    mov ECX, n ; количество повторов
    call sum_n

quit:
    mov eax, 0
    invoke ExitProcess, eax
main endp
end main
```

2.5 Способы передачи параметров в процедуры

Часто в подпрограммы необходимо передавать параметры. Существует три основных места, для передачи параметров:

- через регистры;
- через стек;
- через общие области памяти.

Пусть в подпрограмму testproc нужно передать параметр x.

Пример 8.3 Передача параметров через регистры

```
    mov     EAX, x ; параметр x в регистр AX
    call    testproc     ; вызов процедуры testproc
;----------------------------------------------
testproc proc
    inc     EAX    ; некоторое действие с параметром
    ret            ; возврат из процедуры
testproc endp
;----------------------------------------------
```

```
x dd 0
```

В этом примере для передачи параметра x используется регистр EAX, с которым процедура выполняет некоторое действие и возвращает ответ также через регистр EAX.

Пример 8.4 Передача параметров через стек

```
    push    x       ; параметр X в стек
    call    testproc ; вызов процедуры testproc
;-----------------------------------------------
testproc proc
    pop     EBX     ; временно извлекаем адрес возврата
    pop     EAX     ; извлекаем из стека параметр
    push    EBX     ; возвращаем в стек адрес возврата
    inc     EAX     ; некоторое действие с параметром
    ret             ; возврат из процедуры
testproc endp
;-----------------------------------------------
x    dd 0
```

Здесь параметр x помещается перед вызовом процедуры в стек. В вызванной процедуре testproc происходит извлечение из стека сначала адреса возврата в регистр EBX, а затем самого параметра в регистр EAX. Для правильного возврата из процедуры необходимо вернуть значение адреса возврата в стек, причем так, чтобы во время выполнения команды ret он находился в верхушке стека.

Пример 8.4 Передача параметров через общую область памяти

```
    call    testproc ; вызов процедуры testproc
;-----------------------------------------------
testproc proc
    mov     EAX, x  ; извлекаем параметр
    inc     EAX     ; некоторое действие с параметром
    ret             ; возврат из процедуры
testproc endp
;-----------------------------------------------
x    dd 0
```

При этом способе передачи параметра процедура должна знать, по какому адресу находится значение параметра.

> *Примечание*. Возврат значения из процедуры осуществляется, как правило, через регистр EAX. Это может быть либо сам ответ, либо его адрес.

2.6 Механизмы передачи параметров

Параметры можно передавать с помощью одного из механизмов:

- по значению;
- по ссылке;

- по возвращаемому значению;
- по результату.

2.6.1 Передача параметров по значению

Процедуре передается собственно значение параметра. При этом фактически значение параметра копируется, и процедура использует его копию, так что модификация исходного параметра оказывается невозможной. Этот механизм применяется для передачи небольших параметров, таких как байты или слова.

Например, если параметры передаются в регистрах:

```
mov         ax,word ptr value   ; сделать копию значения
call        procedure           ; вызвать процедуру
```

2.6.2 Передача параметров по ссылке

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

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

В предыдущих примерах к способам передачи параметров подразумевалось, что в процедуру передается значение параметра x. Если же нужно предоставить процедуре возможность самостоятельно извлекать значение параметра, а также, возможно, изменять его значение, то надо передавать в процедуру не значение параметра, а его адрес. Такой способ может быть также полезен при передаче в процедуру массивов, строк или каких-то других блоков данных (например, структур).

Пример 8.5 Передача параметров по ссылке

```
.686
.model flat, stdcall
option casemap :none   ; case sensitive

; Раздел подключения библиотек
    include     \masm32\include\windows.inc
    include     \masm32\include\kernel32.inc
    include     \masm32\include\user32.inc
    includelib  \masm32\lib\kernel32.lib
    includelib  \masm32\lib\user32.lib
; Сегмент данных
.data
n   dd  6               ; кол-во элементов массива X
x   dd 0, 1, 2, 3, 4, 5 ; массив
```

```
; Сегмент кода
.code
sum_array proc
    ; Вычисляет сумму элементов массива
    ; В регистре EBX- адрес массива
    ; В регистре ECX- размер массива
    ; Результат в EAX
    xor EAX, EAX
sum_values:
    add EAX, dword ptr [EBX]    ; Добавляем текущий элемент к сумме
    add EBX, 4                  ; "переходим" на следующий элемент масси
ва
    loop sum_values             ; пока ECX не равен нулю

    ret
sum_array endp

main proc far
    mov     EBX, offset x ; передаем через EBX адрес массива x
    mov     ECX, n        ; передаем через ECX кол-во элементов x
    call    sum_array
quit:
    mov eax, 0
    invoke ExitProcess, eax
main endp
end main
```

2.6.3 Передача параметров по возвращаемому значению

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

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

2.6.4 Передача параметров по результату

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

2.7 Особенности передачи параметров через стек

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

Пример 8.6 Передача параметров через стек

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

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

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

Пример 8.7 Именование параметров

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

```
xyzzy   endp
```

При внимательном анализе этого метода передачи параметров возникает сразу два вопроса: кто должен удалять параметры из стека, процедура или вызывающая ее программа, и в каком порядке помещать параметры в стек. В обоих случаях оказывается, что оба варианта имеют свои «за» и «против», так, например, если стек освобождает процедура (командой RET число_байтов), то код программы получается меньшим, а если за освобождение стека от параметров отвечает вызывающая функция, как в нашем примере, то становится возможным вызвать несколько функций с одними и теми же параметрами просто последовательными командами CALL.

Первый способ, более строгий, используется при реализации процедур в языке Pascal, а второй, дающий больше возможностей для оптимизации, - в языке С. Разумеется, если передача параметров через стек применяется и для возврата результатов работы процедуры, из стека не надо удалять все параметры, но популярные языки высокого уровня не пользуются этим методом. Кроме того, в языке С параметры помещают в стек в обратном порядке (справа налево), так что становятся возможными функции с изменяемым числом параметров (как, например, printf — первый параметр, считываемый из [BP+4], определяет число остальных параметров).

2.8 Локальные переменные

Часто процедурам требуются локальные переменные, которые не будут нужны после того, как процедура закончится. По аналогии с методами передачи параметров можно говорить о локальных переменных в регистрах — каждый регистр, который сохраняют при входе в процедуру и восстанавливают при выходе, фактически играет роль локальной переменной. Единственный недостаток регистров в роли локальных переменных — их слишком мало. Следующий вариант — хранение локальных данных в переменной в сегменте данных — удобен и быстр для большинства несложных ассемблерных программ, но процедуру, использующую этот метод, нельзя вызывать рекурсивно: такая переменная на самом деле является глобальной и находится в одном и том же месте в памяти для каждого вызова процедуры. Третий и наиболее распространенный способ хранения локальных переменных в процедуре — стек. Принято располагать локальные переменные в стеке сразу после сохраненного значения регистра EBP, так что на них можно ссылаться изнутри процедуры, как [EBP-2], [EBP-4], [EBP-6] и т.д.

Пример 8.8 Работа с локальными переменными

```
testproc        proc   near
testproc_x      equ    [EBP+12]    ; параметры
testproc_y      equ    [EBP+8]
```

```
testproc_z          equ    [EBP+4]
testproc_l          equ    [EBP-4]     ; локальные переменные
testproc_m          equ    [EBP-8]
testproc_n          equ    [EBP-12]

        push       EBP           ; сохранить предыдущий BP
        mov        EBP, ESP      ; установить EBP для этой процедуры
        sub        ESP, 6        ; зарезервировать 6 байт для
                                 ; локальных переменных
(тело процедуры)
        mov        ESP, EBP      ; восстановить ESP, выбросив
                                 ; из стека все локальные переменные
        pop        EBP           ; восстановить BP вызвавшей процедуры
        ret        6             ; вернуться, удалив параметры из стека
testproc  endp
```

Внутри процедуры testproc стек будет заполнен следующим образом (см. рисунок 8.1).

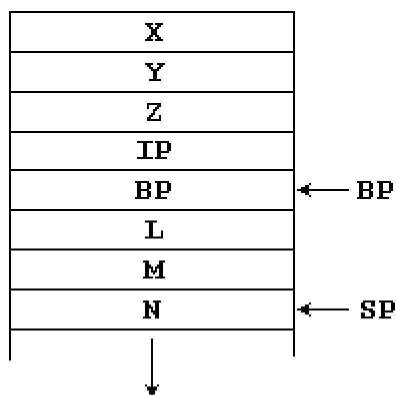


Рисунок 8.1 – Стек при вызове процедуры testproc

Последовательности команд, используемые в начале и в конце таких процедур, оказались настолько часто применяемыми, что в процессоре 80186 были введены специальные команды ENTER и LEAVE, выполняющие эти же самые действия.

Область в стеке, отводимая для локальных переменных вместе с активационной записью, называется стековым кадром.

Пример 8.9 Работа с командами ENTER и LEAVE

```
testproc            proc    near
testproc_x          equ     [bp+8]   ; параметры
testproc_y          equ     [bp+6]
testproc_z          equ     [bp+4]
testproc_l          equ     [bp-2]   ; локальные
testproc_m          equ     [bp-4]   ; переменные
testproc_n          equ     [bp-6]

        enter      6,0              ; push bp
                                    ; mov bp,sp
```

```
                                          ; sub sp,6
;(тело процедуры)
        leave                             ; mov sp,bp
                                          ; pop bp
        ret        6                      ; вернуться,
                                          ; удалив параметры
                                          ; из стека
testproc  endp
```

3 Контрольные вопросы

1) Опишите механизм вызова процедуры.
2) Опишите механизм возврата из процедуры.
3) Чем отличается ближний и дальний вызовы?
4) Опишите форматы команды call.
5) Опишите форматы команды ret.
6) Назовите механизмы передачи параметров в процедуры.
7) Назовите способы передачи параметров в процедуры.
8) Как создать локальные переменные?
9) Для чего приминается команда enter?
10) Как работает команда leave?
11) В чем преимущества передачи параметров через регистры?
12) Как передаются параметры по ссылке?
13) В чем особенность передачи параметров через стек?

4 Задание

1) Выполнить задание в соответствии с пунктом 5.
2) Оформить отчёт.

5 Задание для выполнения работы

Составить три программы (с разными способами передачи параметров в процедуру), содержащую процедуру, выполняющую нижеприведённые действия. Исследовать её работу в отладчике.

1. n - раз выводит на экран символ '*' (в качестве параметра передавать число n).
2. Возводит целое число X в целую степень n (в качестве параметра передавать числа X и n).
3. Проверяет число X на знак и возвращает 0, если число X положительное или 0, и 1 в противном случае (в качестве параметра передавать число X).
4. Проверяет числа X и Y на равенство и возвращает 0, если равны, и 1 в противном случае (в качестве параметра передавать числа X и Y).
5. Вычисляет значение функции |X|, X - целое (в качестве параметра передавать число X).

6. Делит целое число X на n-ю степень числа 2 (в качестве параметра передавать числа X и n).
7. Умножает целое число X на n-ю степень числа 2 (в качестве параметра передавать числа X и n).
8. Преобразует положительное число X в отрицательное (в дополнительный код) (в качестве параметра передавать число X).
9. Проверяет число X на принадлежность диапазону [a, b] и возвращает 0, если принадлежит, и 1 в противном случае (в качестве параметра передавать числа X, a, b).
10. Проверяет число X на принадлежность диапазону [0, a]V[b, 255] и возвращает 0, если принадлежит, и 1 в противном случае (в качестве параметра передавать числа X, a, b).

Практическое занятие № 9. Цепочечные команды

1 Цель работы

Исследовать с помощью отладчика работу команд операций со строками. Научиться использовать цепочечные команды

2 Краткая теория

2.1 Понятие цепочки

Цепочка (строка) – это непрерывная последовательность байт, слов или двойных слов.

Команды работы со строками работают с большими структурами данных в памяти, такими, как алфавитно-цифровые строки символов.

В системе команд процессоров семейства x86 имеется несколько команд, предназначенных для обработки одного элемента строки и автоматического продвижения на следующий элемент. Таким цепочечным командам может предшествовать префикс повторения rep, благодаря которому инициируется повторное выполнение команды над следующим элементом. Благодаря этому цепочки обрабатываются быстрее нежели, чем при организации программного цикла. Количество повторов ограничено максимально возможной длиной цепочки, и может заканчиваться ранее при выполнении определенных условий.

Цепочечные команды не имеют аргументов. В зависимости от семантики команды в качестве источника может выступать либо ячейка памяти, адресуемая парой DS:SI/ESI, либо регистр AL/AX/EAX; в качестве приемника - либо ячейка памяти, адресуемая парой ES:DI/EDI, либо регистр AL/AX/EAX. Таким образом, при необходимости следует инициализировать регистры DS, ES, SI/ESI и DI/EDI.

После выполнения цепочечной команды содержимое регистров SI/ESI и DI/EDI автоматически модифицируется так, чтобы указывать на следующие элементы цепочек. Флаг направления DF определяет увеличивать (DF=0) или уменьшать (DF=1) значение индексного регистра. Величина увеличения/уменьшения зависит от размера элемента и может быть 1, 2 или 4.

Если с цепочечной командой используется префикс повторения, то после каждого ее выполнения происходит декремент CX/ECX. Таким образом CX/ECX следует инициализировать требуемым числом повторений. Когда содержимое счетчика достигает нуля, управление передается следующей по порядку команде.

2.2 Цепочечные команды

2.2.1 Префиксы повторения

Префикс повторения rep имеется несколько вариантов.

- rep - повторять операцию до тех пор, пока CX не равен 0;
- repz или repe - повторять операцию до тех пор, пока флаг ZF = 0;
- repnz или repne - повторять операцию до тех пор, пока флаг ZF не равен 0.

Префиксы повторения rep (Повторять, пока ECX не равен нулю), repe/repz (Повторять пока равно/ноль) и repne/repnz (Повторять пока не равно/не ноль) задают повторяющееся выполнение команд работы со строками. Эта форма итераций позволяет операциям работы со строками работать быстрее, чем это возможно при организации программных циклов.

Когда у команды работы со строкой имеется префикс повторения, операция выполняется до тех пор, пока одно из условий окончания, определяемое префиксом, не будет выполнено.

Для каждого повторения команды работа со строкой может быть приостановлена прерыванием или исключением. После того, как прерывание или исключение было обработано, операция работы над строкой может быть продолжена с того места, на котором она была приостановлена. Этот механизм позволяет выполнять длинные операции над строками без влияния на время отклика системы на прерывание.

Префиксы повторения отличаются друг от друга по своим дополнительным условиям окончания работы. Префикс rep не имеет дополнительных условий окончания. Префиксы repe/repz и repne/repnz используются исключительно с командами scas (Сканировать строку) и cmps (Сравнить строки). Префикс repe/repz заканчивает работу, если флаг ZF очищен. Префикс repne/repnz заканчивает работу, если флаг ZF установлен. Флаг ZF не требует предварительной инициализации перед выполнением повторяющихся команд работы со строками, так как и scas и cmps воздействуют на флаг ZF в соответствие с результатами сравнения, которое они выполнили.

2.2.2 Команды обработки цепочек

Для обработки цепочек микропроцессор имеет следующие команды:
- movs - переслать элемент одной цепочки в другую;
- lods - загрузить элемент цепочки в аккумулятор;
- stos - сохранить содержимое аккумулятора в элементе цепочки;
- cmps - сравнить содержимое элементов двух цепочек;
- scas - сравнить содержимое аккумулятора с элементом цепочки.

Хотя регистры общего назначения в большинстве случаев полностью взаимозаменяемы, команды работы со строками требуют использования двух специальных регистров. Строки источника и преемника находятся в памяти, адресуемой регистрами ESI и EDI. Регистр ESI указывает на операнд-источник. По умолчанию, регистр ESI используется вместе с сегментным регистром DS. Префикс замены сегмента позволяет использовать регистр ESI вместе с сегментными регистрами CS, SS, ES, FS

или GS. Регистр EDI указывает на операнд-преемник. Он использует сегмент, на который указывает сегментный регистр ES; замена этого сегмента запрещена. Использование двух различных сегментных регистров в одной команде позволяет работать со строками, расположенными в различных сегментах.

Когда в командах работы со строками используются регистры ESI и EDI, они автоматически увеличиваются или уменьшаются после каждой итерации. Действия над строками можно начинать выполнять со старших адресов по направлению к младшим адресам, или они могут выполняться, начиная с младших адресов по направлению к старшим адресам. Направление операций управляется флагом DF.

Если флаг очищен, значения регистров увеличиваются. Если флаг установлен, значения регистров уменьшаются. Команды std и cld устанавливают и очищают этот флаг. Программисты всегда должны поместить нужное значение во флаг DF, прежде чем использовать команды работы со строками.

```
movs DST, SRC
```

Переслать строку перемещает элемент строки, адресуемый регистром ESI в позицию, адрес которой указан в регистре EDI. Команда movsb перемещает байты, команда movsw перемещает слова и команда movsd перемещает двойные слова. Команда movs, когда она дополнена префиксом rep, работает как пересылка блоков из памяти в память. Для выполнения этой операции программа должна инициализировать регистры ECX, ESI и EDI. Регистр ECX указывает количество элементов в блоке.

```
cmps DST, SRC
```

Сравнение строк вычитает элемент строки-преемника из элемента строки-источника и обновляет флаги AF, SF, PF, CF и OF. Ни один из элементов не будет записан обратно в память. Если элементы строки равны, устанавливается флаг ZF; в противном случае этот флаг очищается. cmpsb сравнивает байты, cmpsw сравнивает слова и cmpsd сравнивает двойные слова.

```
scas DST
```

Сканировать строку вычитает элемент строки-преемника из регистра EAX, AX или AL (в зависимости от длины операнда) и обновляет флаги AF, SF, ZF, PF и OF. Строка и регистры не изменяются. Если значения равны, устанавливается флаг ZF; в противном случае флаг очищается. Команда scasb сканирует байты, команда scasw сканирует слова и команда scasd сканирует двойные слова.

Когда префиксы repe/repz или repne/repnz модифицируют команду scas или cmps, сформированный при этом цикл заканчивается либо по

счетчику цикла, либо по тому воздействию, которое команды scas или cmps оказывают на флаг ZF.

```
lods SRC
```

Загрузить строку помещает элемент строки-источника, на который указывает адрес в регистре ESI, в регистр EAX для строк из двойных слов, в регистр AX для строк из слов или в регистр AL для строк из байтов. Эта команда используется обычно внутри цикла, где другие команды обрабатывают каждый элемент строки по мере его появления в регистре.

```
stos DST
```

Запомнить строку помещает элемент строки из регистра EAX, AX или AL в строку, на которую указывает адрес в регистре EDI. Эта команда обычно используется в цикле, где она записывает в память результат обработки элемента строки, считанного из памяти при помощи команды lods. Команда rep stos является самым быстрым способом инициализации большого блока памяти.

Схема функционирования цепочечных коанд показана на рисунке 9.1.

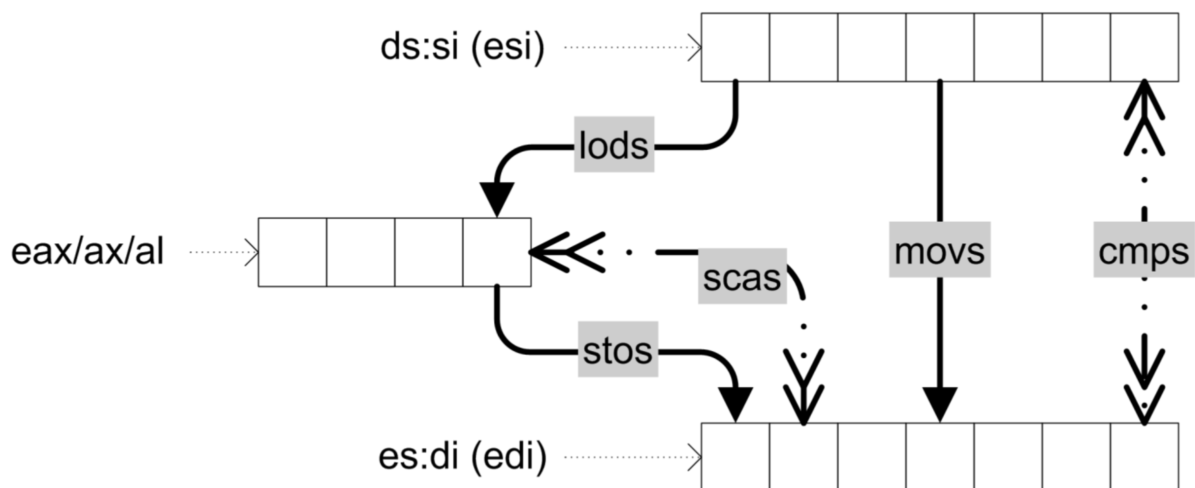


Рисунок 9.1 – Цепоченые команды

Пример 9.1. Копирования строки

```
lea     ESI, src ; адрес источника
lea     EDI, dst ; адрес приемника
mov     ECX, 10  ; длина копируемой строки
rep     movsb    ; копирование
;-------------------------------------------
src     db "0123456789"
dst     db 10 dup(" ")
```

Пример 9.2 Изменения регистра латинской буквы

```
lea     ESI, src ; адрес источника
lea     EDI, dst ; адрес приемника

mov ECX, 6

next:   lodsb   ; загрузить символ в AL
        cmp AL, "a"
        jb no_modify
        and AL, 11011111b
no_modify:      stosb
                loop next
;-----------------------------------------
src     db "abcABC"
dst     db 6 dup(" ")
```

Пример 9.3 Поиск и замена символа

```
lea     ESI, src ; адрес источника
lea     EDI, src ; адрес приемника

mov     ECX, 5

mov     AL, "x"

next:   repne scasb     ; сравнить символ с AL
        jnz no_modify
        dec EDI
        mov byte ptr[EDI], 20h; заменить на пробел

no_modify:      jcxz done
                jmp next
        ;
done:
;------------------------------------------------
src     db "1x2x3"
```

Пример 9.4 Поиск подстроки

```
lea     ESI, src ; адрес источника
lea     EDI, dst ; адрес приемника

mov     ECX, 9
mov     AL, byte ptr[ESI]

        repne scasb     ; найти начальный символ
        jne no          ; если начальный символ не найден
        mov     ECX, 3
        dec EDI
        rep cmpsb       ; сравнить с подстрокой
        je yes
        jmp no
        ;

yes:    ; Found
```

```
        mov     EAX, 1
        jmp     done
        ;
no:     ; Not Found
        mov     EAX, 0
        ;
done:
;--------------------------------------------------------
src     db "123"
dst     db "abc123abc"
```

2.3 Виды строк

Следует различать три разных вида строк:

- символьные массивы (длина строки фиксирована);
- паскаль-строки (строки ограниченной длины, длина хранится в нулевом байте символьного массива);
- си-строки (строки ограничиваются нулевым символом).

Пример 9.5 Объявление различных видов строк

```
; Символьный массив
Char_Array db 'ABC', 17 dup(?)
; Паскаль-строка (максимальная длина 20)
Pascal_String db 3, 'ABC', 17 dup(?)
; Си-строка (17-ть байтов - буфер(необязательно))
C_String db 'ABC', 0, 17 dup(?)
```

2.4 Трансляция в соответствии с таблицей

```
xlat адрес_таблицы_байтов
xlatb
```

Помещает в AL байт из таблицы в памяти по адресу ES:BX (или ES:EBX) со смещением относительно начала таблицы, равным AL. В качестве аргумента для XLAT в ассемблере можно указать имя таблицы, но эта информация никак не используется процессором и служит только как комментарий. Если этот комментарий не нужен, можно применить форму записи XLATB. В качестве примера использования XLAT можно написать следующий вариант преобразования шестнадцатеричного числа в ASCII-код соответствующего ему символа:

```
mov     al,0Ch
mov     bx, offset htable
xlatb
```

если в сегменте данных, на который указывает регистр ES, было записано

```
htable  db     "0123456789ABCDEF"
```

то теперь AL содержит не число 0Ch, а ASCII-код буквы «C». Разумеется, это преобразование можно выполнить, используя гораздо более компактный код всего из трех арифметических команд, но с XLAT можно выполнять любые преобразования такого рода.

3 Контрольные вопросы

1) Что такое цепочки (строки)?
2) Какие команды для работы со строками вы знаете?
3) Синтаксис и семантика команд цепочечных команд.
4) Какие префиксы повторения вы знаете?
5) Какие команды управления направлением обработки цепочек вы знаете?

4 Задание

1) Выполнить задание в соответствии с пунктом 5.
2) Оформить отчёт.

5 Задание для выполнения работы

Составить программу, содержащую процедуру (передача параметров в процедуру через стек), выполняющую нижеприведённые действия.

1. Подсчитывает кол-во слов в Си-строке (слова разделяются произвольным кол-вом пробелов).
2. Преобразует строчные латинские буквы в Си-строке в прописные (в строке могут быть не только буквы).
3. Сравнивает две Си-строки на равенство (строки считаются равными, если они имеют одинаковую длину и побайтно совпадают).
4. Очищает данную Си-строку (заполняет её нулями).
5. Вставляет одну Си-строку в заданное место в другой Си-строки.
6. Объединяет две Си-строки (результат объединения записывается в третью Си-строку).
7. Определяет кол-во вхождений одной Си-строки в другую Си-строку.
8. Подсчитывает кол-во слов в Паскаль-строке (слова разделяются произвольным кол-вом пробелов).
9. Преобразует строчные латинские буквы в Паскаль-строке в прописные (в строке могут быть не только буквы).
10. Сравнивает две Паскаль-строки на равенство (строки считаются равными, если они имеют одинаковую длину и побайтно совпадают).
11. Очищает данную Паскаль-строку (заполняет её нулями).
12. Вставляет одну Паскаль-строку в заданное место в другой Паскаль-строки.
13. Объединяет две Паскаль-строки (результат объединения записывается в третью Паскаль-строку).

14. Определяет колво вхождений одной Паскаль-строки в другую Паскаль-строку.
15. Подсчитывает кол-во слов в символьном массиве (слова разделяются произвольным кол-вом пробелов).
16. Преобразует строчные латинские буквы в символьном массиве в прописные (в строке могут быть не только буквы).
17. Сравнивает два символьных массива на равенство (строки считаются равными, если они имеют одинаковую длину и побайтно совпадают).
18. Очищает данный символьный массив (заполняет его нулями).
19. Вставляет содержимое одного символьного массива в заданное место другого символьного массива.
20. Объединяет два символьных массива (результат объединения записывается в третий символьный массив).
21. Определяет кол-во вхождений содержимого одного символьного массива в другой символьный массив.

Список рекомендуемой литературы

1 Аблязов, Р. З. Программирование на ассемблере на платформе x86-64 [Электронный ресурс] / Р. З. Аблязов. — Электрон. текстовые данные. — Саратов : Профобразование, 2017. — 304 c. — 978-5-4488-0117-4. — Режим доступа: http://www.iprbookshop.ru/63951.html

2 Гуров, В.В. Архитектура микропроцессоров : учебное пособие / В.В. Гуров. — 2-е изд. — Москва : ИНТУИТ, 2016. — 327 с. — ISBN 978-5-9963-0267-3. — Текст : электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/100570

3 Иванова, Г.С. Основные приемы программирования на ассемблере MASM32 : учебное пособие / Г.С. Иванова, Т.Н. Ничушкина. — Москва : МГТУ им. Н.Э. Баумана, 2016. — 56 с. — ISBN 978-5-7038-4455-7. — Текст : электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/103558

4 Куляс, О.Л. Курс программирования на ASSEMBLER : учебное пособие / О.Л. Куляс, К.А. Никитин. — Москва : СОЛОН-Пресс, 2017. — 220 с. — ISBN 978-5-91359-245-3. — Текст : электронный // Электронно-библиотечная система «Лань» : [сайт]. — URL: https://e.lanbook.com/book/107672

5 Микропроцессорные системы: Учебное пособие/Гуров В.В. - М.: НИЦ ИНФРА-М, 2016. - 336 с.: 60x90 1/16. - (Высшее образование: Бакалавриат) (Переплёт) ISBN 978-5-16-009950-7, 500 экз. http://znanium.com/bookread2.php?book=462986

6 Юров В.И. Assembler [Текст] : [учеб. для вузов] / В. И. Юров. - 2-е изд. - СПб. : Питер, 2011 (51135). - 636 с. : ил., табл. - (Учеб. для вузов). - Библиогр.: с. 625 (18 назв.). - ISBN 978-5-94723-581-4.

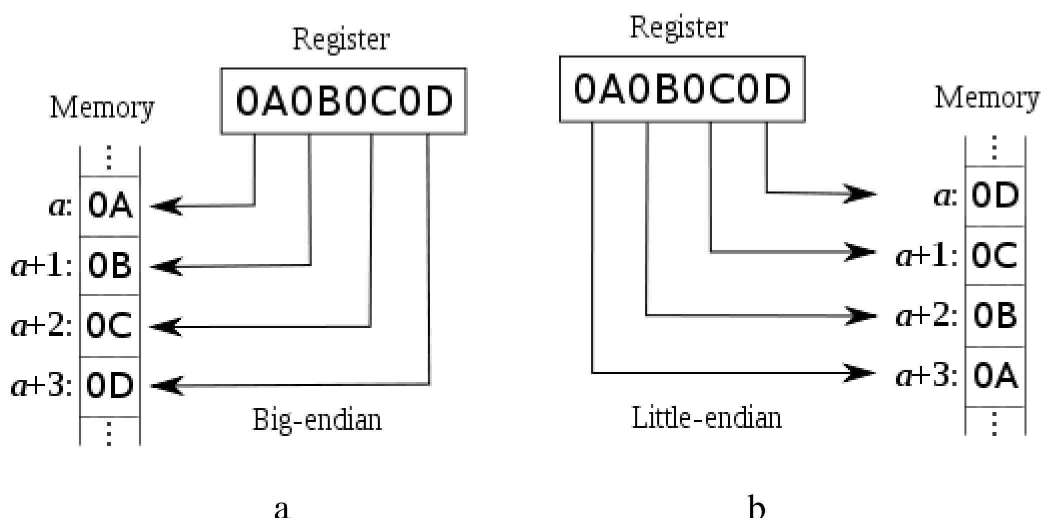
Приложение А
(справочное)
Организация памяти

Байт является минимально адресуемой единицей информации в оперативной памяти. При этом следует учитывать порядок расположения байт в регистрах и хранение этих байт в памяти. Выделяют два порядка хранения байт в памяти:

- Порядок от старшего к младшему (big-endian);
- Порядок от младшего к старшему (little-endian).

Порядок от старшего к младшему или big-endian: запись начинается со старшего и заканчивается младшим байтом. Этот порядок является стандартным для протоколов TCP/IP (поэтому его часто называют сетевым порядком байтов) и используется процессорами IBM 360/370/390, Motorola 68000, SPARC (рисунок А.1, а). В этом же виде (используя представление в десятичной системе счисления) записываются числа индийско-арабскими цифрами в письменностях с порядком знаков слева направо.

Порядок от младшего к старшему или little-endian: запись начинается с младшего и заканчивается старшим байтом. Этот порядок записи принят в памяти персональных компьютеров с x86-процессорами, в связи с чем иногда его называют интеловский порядок байтов (рисунок А.1, b).



a b

Рисунок А.1 – Различия порядка байт big-endian и little-endian

Примеры записи чисел в памяти представлены на рисунке А.2.

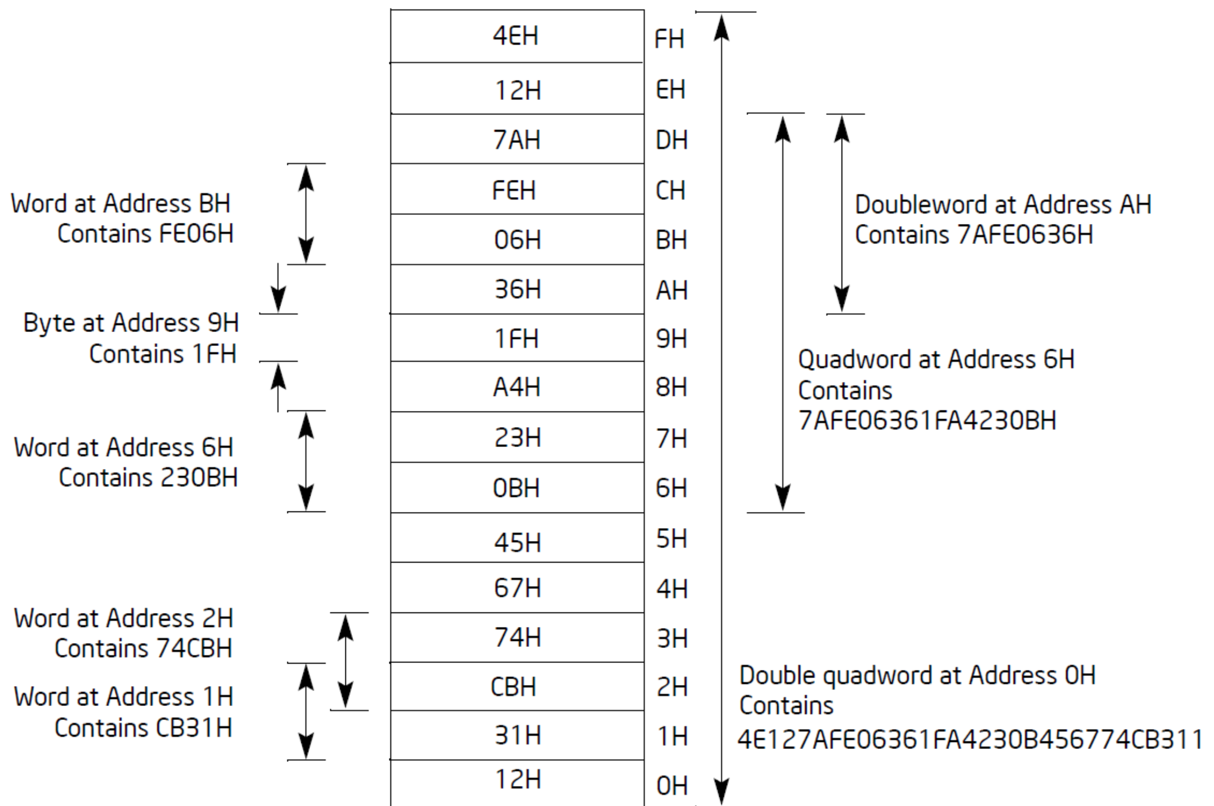


Рисунок А.2 – Примеры записи чисел в памяти

Приложение Б
(справочное)
Содержимое регистра флагов

Регистр флагов (EFLAGS/FLAGS) определяют текущее состояние машины и содержат результаты выполнения некоторых команд (многие арифметические команды и команды сравнения изменяют состояние флагов) (рисунок Б.1).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

Reserved bit positions. DO NOT USE.
Always set to values previously read.

Рисунок Б.1 – Содержимое регистра eflags

Рассмотрим основные флаги (Таблица Б.1).

Таблица Б.1 – Назначение флагов регистра eflags

Флаг	**Название**	**№ бита**	**Назначение**
Флаги состояния			
cf	Флаг переноса Carry Flag	0	Содержит результат переноса из старшего бита после арифметических операций. 1 - Арифметическая операция произвела перенос из старшего бита результата. Старшим является 7, 15 или 31-й бит в зависимости от размерности операнда;

			0 - переноса не было.
pf	Флаг четности Parity Flag	2	Содержит 1, если в младших 8-ми битах результата четное количество единиц, и 0 - в противном случае. Этот флаг — только для 8 младших разрядов операнда любого размера.
af	Вспомогательный флаг переноса Auxiliary carry flag	4	Только для команд работающих с BCD-числами. Содержит результат переноса из 3-го бита в байте при операциях с целыми числами в BCD-формате. Фиксирует факт заема из младшей тетрады результата: 1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; 0 — переносов и заемов в/из 3 разряд(а) младшей тетрады результата не было
zf	Флаг нуля Zero Flag	6	Содержит 1, если результат последней арифметической операции равен нулю, и 0 – в противном случае.
sf	Флаг знака Sign Flag	7	Содержит знак результата последней арифметической операции (0 - плюс, 1 - минус). Отражает состояние старшего (знакового) бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно).
of	Флаг переполнения Overflow Flag	11	Ситуация переполнения возникает, когда результат умножения или деления не помещается в приемнике. Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 — в результате операции происходит перенос (заем) в/из старшего, знакового бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно);

			0 — в результате операции не происходит переноса (займа) в/из старшего, знакового бита результата
iopl	Уровень Привилегий ввода-вывода (Input/Output Privilege Level)	12-13	Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода в зависимости от привилегированности задачи
nt	Флаг вложенности задачи Nested Task	14	Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую
Флаг управления			
df	Флаг направления Direction flag	10	Указывает на левое или правое направление при операциях со строками.
Системные флаги			
tf	Флаг трассировки Trace Flag	8	Предназначен для организации пошаговой работы микропроцессора. 1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 — обычная работа
if	Флаг прерывания Interrupt enable Flag	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR). 1 — аппаратные прерывания разрешены; 0 — аппаратные прерывания запрещены
rf	Флаг возобновления Resume Flag	16	Используется при обработке прерываний от регистров отладки.
Vm	Флаг виртуального 8086 Virtual 8086 Mode	17	Признак работы микропроцессора в режиме виртуального 8086. 1 — процессор работает в режиме виртуального 8086; 0 — процессор работает в реальном или защищенном режиме

ac	Флаг контроля выравнивания (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом am в системном регистре cr0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некратным адресам будут возбуждать исключительную ситуацию

Приложение В
(справочное)
Команды безусловного перехода

Безусловные переходы осуществляются с помощью команды jmp, которая может использоваться в 5 разновидностях. Переход может быть:

- прямым коротким (в пределах -128... + 127 байтов);
- прямым ближним (в пределах текущего сегмента команд):
- прямым дальним (в другой сегмент команд);
- косвенным ближним (в пределах текущего сегмента команд через ячейку с адресом перехода);
- косвенным дальним (в другой сегмент команд через ячейку с адресом перехода).

Рассмотрим последовательно структуру программ с переходами разного вида.

В.1 Прямой короткий (short) переход

Прямым называется переход, в команде которого в явной форме указывается метка, на которую нужно перейти. Разумеется, эта метка должна присутствовать в том же программном сегменте, при этом помеченная ею команда может находиться как до, так и после команды jmp. Достоинство команды короткого перехода заключается в том, что она занимает лишь 2 байт памяти: в первом байте записывается код операции (EBh), во втором – смещение к точке перехода. Расстояние до точки перехода отсчитывается от очередной команды, т.е. команды, следующей за командой jmp. Поскольку требуется обеспечить переход как вперед, так и назад, смещение рассматривается, как число со знаком и, следовательно, переход может быть осуществлен максимум на 127 байт вперед или 128 байт назад. Прямой короткий переход оформляется следующим образом:

```
.code
…
jmp short go ;Код EB dd

…

go:
…
code ends
```

В комментарии указан код команды; dd (от displacement, смещение) обозначает байт со смещением к точке перехода от команды, следующей за командой jmp. При выполнении команды прямого короткого перехода процессор прибавляет значение байта dd к младшему байту текущего значения указателя команд IP (который, всегда указывает на команду, следующую за выполняемой). В результате в IP оказывается адрес точки

перехода, а предложения, находящиеся между командой jmp и точкой перехода, не выполняются.

> *Примечание*. Конструкция с прямым переходом вперед часто используется для того, чтобы обойти данные, которые по каким-то причинам желательно разместить в сегменте команд.

В.2 Прямой ближний (near) переход

Прямой ближний (near) (или внутрисегментный) переход отличается от предыдущего только тем, что под смещение к точке перехода отводится целое слово (16 бит).

```
code segment
…
jmp go ;Код E9 dddd
…
go:
…
code ends
```

> *Примечание.* В 64-битном режиме работы процессора используется не 16-битное, а 32-битное смещение.

Метка go может находиться в любом месте сегмента команд, как до, так и после команды jmp. В коде команды dddd обозначает слово с величиной относительного смещения к точке перехода от команды, следующей за командой jmp.

При выполнении команды прямого ближнего перехода процессор должен прибавить значение слова dddd к текущему значению указателя команд IP и сформировать тем самым адрес точки перехода.

В.3 Прямой дальний (far) переход

Прямой дальний (far) (или межсегментный) переход позволяет передать управление в любую точку любого сегмента. При этом предполагается, что программа включает несколько сегментов команд. Команда дальнего перехода включает, кроме кода операции EAh, еще и полный адрес точки перехода, т.е. сегментный адрес и смещение. Транслятору надо сообщить, что этот переход – дальний (по умолчанию команда jmp транслируется, как команда ближнего перехода). Это делается с помощью описателя far ptr, указываемого перед именем точки перехода.

```
code1 segment

assume CS: code1 ;Сообщим транслятору, что это сегмент команд
…
jmp far ptr go ;Код EA dddd ssssы
…
```

```
code1 ends


code2 segment

assume CS : code2 ;Сообщим транслятору, что это сегмент команд
…
go:
…
code2 ends
```

Метка go находится в другом сегменте команд этой двухсегментной программы. В коде команды ssss – сегментный адрес сегмента code2, а dddd – смещение точки перехода go в сегменте команд code2.

Все виды прямых переходов требуют указания в качестве точки перехода программной метки. С одной стороны, это весьма наглядно; просматривая текст программы, можно сразу определить, куда осуществляется переход. С другой стороны, такой переход носит статический характер – его нельзя настраивать по ходу программы. Еще более серьезный недостаток прямых переходов заключается в том, что они не дают возможность перейти по известному абсолютному адресу, т.е. не позволяют обратиться ни к системным средствам, ни вообще к другим загруженным в память программам. Действительно, программы операционной системы не имеют никаких меток, так как метка – это атрибут исходного текста программы, а программы операционной системы транслировались не нами и присутствуют в компьютере только в виде выполнимых модулей. А вот адреса каких-то характерных точек системных программ определить можно. Для обращения по абсолютным адресам надо воспользоваться командами косвенных переходов, которые, как и прямые, могут быть ближними и дальними.

В.4 Косвенный ближний переход

В отличие от команд прямых переходов, команды косвенных переходов могут использовать различные способы адресации и, соответственно, иметь много разных вариантов. Общим для них является то, что адрес перехода не указывается явным образом в виде метки, а содержится либо в ячейке памяти, либо в одном из регистров. Это позволяет при необходимости модифицировать адрес перехода, а также осуществлять переход по известному абсолютному адресу. Рассмотрим случай, когда адрес перехода хранится в ячейке сегмента данных. Если переход ближний, то ячейка с адресом состоит из одного слова и содержит только смещение к точке перехода.

```
; Сегмент данных
.data
jump_address    dd go ;Адрес перехода
```

```
; Сегмент кода
.code
main:

    jmp DS:jump_address     ;Код FF 26 dddd

...

    go: ;Точка перехода

...

quit:
```

Точка перехода go может находиться в любом месте сегмента команд. В коде команды dddd обозначает относительный адрес слова jump_address в сегменте данных, содержащем эту ячейку.

В приведенном фрагменте адрес точки перехода в слове jump_address задан однозначно указанием имени метки go. Такой вариант косвенного перехода выполняет фактически те же функции, что и прямой (переход по единственному заданному заранее адресу), только несколько более запутанным образом. Достоинства косвенного перехода будут более наглядны, если представить, что ячейка jump_address поначалу пуста, а по ходу выполнения программы внес, в зависимости от каких-либо условий, помещается адрес той или иной точки перехода:

```
    mov jump_address, offset go1
...
    mov jump_address, offset go2
...
    mov jump_address, offset go3
```

Разумеется, приведенные выше команды должны выполняться не друг за другом, а альтернативно. В этом случае создается возможность перед выполнением перехода определить или даже вычислить адрес перехода, требуемый в данных условиях.

Ассемблер допускает различные формы описания косвенного перехода через ячейку сегмента данных:

```
    jmp DS:jump_address
    jmp dword ptr jump_address
    jmp jump_address
```

В первом варианте, использованном в приведенном выше фрагменте, указано, через какой сегментный регистр надлежит обращаться к ячейке jump_address, содержащей адрес перехода. Здесь допустима замена сегмента, если сегмент с ячейкой jump_address адресуется через другой сегментный регистр, например, ES или CS.

Во втором варианте подчеркивается, что переход осуществляется через ячейку размером в одно слово и, следовательно, является ближним. Ячейка jump_address объявлена с помощью директивы dd и содержит двухсловный адрес перехода, требуемый для реализации перехода. Описатель dword ptr перед именем ячейки с адресом перехода заставляет транслятор считать, что она имеет размер 2 слова (независимо от того, как она была объявлена).

Наконец, возможен и самый простой, третий вариант, который совпадает по форме с прямым переходом, но, тем не менее, является косвенным, так как символическое обозначение jump_address является именем поля данных, а не программной меткой. В этом варианте предполагается, что сегмент, в котором находится ячейка jump_address, адресуется по умолчанию через регистр DS, хотя, как и во всех таких случаях, допустима замена сегмента. Тип перехода (ближний или дальний) определяется, исходя из размера ячейки jump_address. Однако этот вариант не всегда возможен. Для его правильной трансляции необходимо, чтобы транслятору к моменту обработки предложения

```
jmp jump_address
```

было уже известно, что собой представляет имя jump_address. Этого можно добиться двумя способами. Первый - расположить сегмент данных до сегмента команд, а не после, как в приведенном выше примере. Второй - заставить транслятор обрабатывать исходный текст программы не один раз, как он это делает по умолчанию, а несколько.

В приведенных примерах адрес поля памяти с адресом точки перехода задавался непосредственно в коде команды косвенного перехода. Однако этот адрес можно задать и в одном из регистров общего назначения (EBX, ESI или EDI). Для приведенного выше примера косвенного перехода в точку go, адрес которой находится в ячейке jump_address в сегменте данных, переход с использованием косвенной регистровой адресации может выглядеть следующим образом:

```
mov EBX, offset jump_address ;В EBX смещение поля с адресом перехода
jmp dword ptr [EBX] ;Переход в точку go
```

Особенно большие возможности предоставляет методика косвенного перехода с использованием базово-индексной адресации через пары регистров, например, [EBX][ESI] или [EBX][EDI]. Этот способ удобен в тех случаях, когда имеется ряд альтернативных точек перехода, выбор которых зависит от некоторых условий. В этом случае в сегменте данных создается не одно поле с адресом, а таблица адресов переходов. В базовый регистр EBX загружается адрес этой таблицы, а в один из индексных регистров – определенный тем или иным способом индекс в этой таблице. Переход

осуществляется в точку, соответствующую заданному индексу. Структура программы, использующей такую методику, выглядит следующим образом:

```
; Сегмент данных
.data
jmp_tbl  label word ;Таблица адресов переходов
go1_addr dd go1 ;Адрес первой альтернативной точки перехода
go2_addr dd go2 ;Адрес второй альтернативной точки перехода
go3_addr dd go3 ;Адрес третьей альтернативной точки перехода

; Сегмент кода
.code
main:

    lea EBX, jmp_tbl ;Загружаем в EBX базовый адрес таблицы
    mov ESI, 4       ;Вычисленное каким-то образом смещение в таблице
    jmp dword ptr [EBX][ESI] ;Если индекс = 4, переход в точку go2

go1: ;1-я точка перехода
...
go2: ;2-я точка перехода
...
go3: ;3-я точка перехода
...
```

Приведенный пример носит условный характер; в реальной программе индекс, помещаемый в регистр ESI, должен вычисляться по результатам анализа некоторых условий.

Наконец, существует еще одна разновидность косвенного перехода, в котором не используется сегмент данных, а адрес перехода помещается непосредственно в один из регистров общего назначения. Часто такой переход относят к категории прямых, а не косвенных, однако это вопрос не столько принципа, сколько терминологии.

Применительно к обозначениям последнего примера такой переход будет выглядеть, например, следующим образом:

```
    mov EBX, offset go1
    jmp EBX
```

Здесь, как и в предыдущих вариантах, имеется возможность вычисления адреса перехода, однако нельзя этот адрес индексировать.

В.5 Косвенный дальний (межсегментный) переход

Как и в случае ближнего перехода, переход осуществляется по адресу, который содержится в ячейке памяти, однако эта ячейка содержит полный (сегмент плюс смещение) адрес точки перехода. Программа в этом случае должна включать, по меньшей мере, два сегмента команд.

Как и в случае ближнего косвенного перехода, ассемблер допускает различные формы описания дальнего косвенного перехода через ячейку сегмента данных:

```
jmp DS:jmp_addr ;Возможна замена сегмента
jmp fword ptr jmp_addr ;Если поле jmp_addr объявлено операторами df
jmp jmp_addr ;Характеристики ячейки должны быть известны
```

Для дальнего косвенного перехода, как и для ближнего, допустима адресация через регистр общего назначения, если в него поместить адрес поля с адресом перехода:

```
mov EBX,offset jmp_addr
jmp fword ptr [EBX]
```

Возможно также использование базово-индексной адресации, если в сегменте данных имеется таблица с двухсловными адресами точек переходов.

В.6 Явление оборачивания адреса

Рассматривая вычисление адреса точки перехода, следует иметь в виду явление оборачивания, суть которого можно кратко выразить такими соотношениями:

```
FFFFFFFFh + 00000001h = 00000000h
00000000h - 00000001h = FFFFFFFFh
```

Если последовательно увеличивать содержимое какого-либо регистра или ячейки памяти, то, достигнув верхнего возможного предела 0xFFFFFFFF, число "перевалит" через эту границу, станет равным нулю и продолжит нарастать в области малых положительных чисел (1, 2, 3, и т.д.). Точно так же, если последовательно уменьшать некоторое положительное число, то оно, достигнув нуля, перейдет в область отрицательных (или, что то же самое, больших беззнаковых) чисел, проходя значения 2, 1, 0, FFFFFFFFh, FFFFFFFEh и т.д.

Таким образом, при вычислении адреса точки перехода смещение следует считать числом без знака, но при этом учитывать явление оборачивания. Если команда jmp находится где-то в начале сегмента команд, а смещение имеет величину порядка 2**32, то переход произойдет вперед, к концу сегмента. Если же команда находится в конце сегмента команд, а смещение имеет ту же величину порядка 2**32, то для определения точки перехода надо двигаться по сегменту вперед, дойти до его конца и продолжать перемещаться от начала сегмента по-прежнему вперед, пока не будет пройдено заданное в смещении число байтов. Для указанных условий мы попадем в точку, находящуюся недалеко от команды jmp со стороны меньших адресов.

НИЗКОУРОВНЕВОЕ ПРОГРАММИРОВАНИЕ

Методические указания к практическим занятиям для студентов всех форм обучения направления подготовки
09.03.04 Программная инженерия

Составители: Мурлин Алексей Георгиевич
Волик Александр Георгиевич
Симоненко Евгений Анатольевич

Авторская правка

Компьютерная вёрстка А. Г. Мурлин