Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

Д. А. ЯКУБОВИЧ Ю. А. МЕДВЕДЕВ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕР. MACRO ASSEMBLER

Практикум

Владимир 2017

УДК 4.438=93.Ассемблер
ББК 32.973.2
Я49

Рецензенты:
Кандидат педагогических наук, доцент
зав. кафедрой информатизации образования
Владимирского института развития образования имени Л. И. Новиковой
В. А. Полякова

Доктор физико-математических наук
профессор кафедры математического анализа
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
Ю. А. Алхутов

Печатается по решению редакционно-издательского совета ВлГУ

Я49 **Якубович, Д. А.** Программирование на языке ассемблер. Macro Assembler : практикум / Д. А. Якубович, Ю. А. Медведев ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2017. – 191 с.
ISBN 978-5-9984-0774-1

Формирует основные базовые и специфические знания, умения и навыки по низкоуровневому программированию на ассемблере. Включает 22 занятия по основам архитектуры процессоров семейства Intel x86, программирования 16-битных и 32-битных приложений. Раскрыты принципы организации адресации ОЗУ, регистрового устройства процессора, использование процедур и макросов. Приведены задания и вопросы для самостоятельной работы и самопроверки.

Предназначен для студентов направлений 44.03.05 – Педагогическое образование по профилям «Математика. Информатика», «Информатика. Математика». Может быть рекомендован студентам других профильных специальностей в качестве основного или дополнительного литературного источника.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Библиогр.: 9 назв.

УДК 4.438=93.Ассемблер
ББК 32.973.2

ISBN 978-5-9984-0774-1 ©ВлГУ, 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 7
1. Цели освоения дисциплины 8
2. Место дисциплины в структуре основной образовательной программы ВПО 9
3. Компетенции обучающегося, формируемые в результате освоения дисциплины (модуля) 9

Часть 1. АРХИТЕКТУРА ПРОЦЕССОРОВ INTEL x86 11
Практическое занятие № 1.1. ЯЗЫК АССЕМБЛЕРА. ОБЗОР АССЕМБЛЕРОВ 11
1. Цели работы 11
2. Теоретический блок 11
3. Вопросы для самопроверки 15
Практическое занятие № 1.2. ПРЕДСТАВЛЕНИЕ ЧИСЕЛ В КОМПЬЮТЕРЕ. МОДЕЛЬ ПАМЯТИ 16
1. Цели работы 16
2. Теоретический блок 16
3. Вопросы для самопроверки 21
4. Самостоятельная работа 21
Практическое занятие № 1.3. ПРОЦЕССОРЫ СЕМЕЙСТВА Intel x86. РЕЖИМЫ АДРЕСАЦИИ 22
1. Цели работы 22
2. Теоретический блок 22
3. Вопросы для самопроверки 28
4. Самостоятельная работа 28
Практическое занятие № 1.4. РЕГИСТРЫ ПРОЦЕССОРА. ОПЕРАТИВНАЯ ПАМЯТЬ 28
1. Цели работы 28
2. Теоретический блок 28
3. Вопросы для самопроверки 34
4. Самостоятельная работа 34

Практическое занятие № 1.5. СБОРКА АССЕМБЛЕРНОЙ ПРОГРАММЫ ... 35
1. Цели работы ... 35
2. Теоретический блок ... 35
3. Вопросы для самопроверки ... 43
4. Самостоятельная работа ... 43

Часть 2. MACRO ASSEMBLER: IA-16 ... 44
Практическое занятие № 2.1. ОСНОВЫ MASM. ТИПЫ ДАННЫХ. КОМАНДЫ И ДИРЕКТИВЫ ... 44
1. Цели работы ... 44
2. Теоретический блок ... 44
3. Вопросы для самопроверки ... 52
4. Самостоятельная работа ... 53
Практическое занятие № 2.2. ШАБЛОН КОНСОЛЬНОГО ПРИЛОЖЕНИЯ. ЗНАКОМСТВО С ОТЛАДЧИКОМ ... 53
1. Цели работы ... 53
2. Теоретический блок ... 54
3. Вопросы для самопроверки ... 59
4. Самостоятельная работа ... 59
Практическое занятие № 2.3. КОМАНДЫ ПЕРЕСЫЛКИ ДАННЫХ И АРИФМЕТИЧЕСКИЕ КОМАНДЫ ... 60
1. Цель работы ... 60
2. Теоретический блок ... 60
3. Вопросы для самопроверки ... 68
4. Самостоятельная работа ... 68
Практическое занятие № 2.4. КОМАНДЫ УСЛОВНОГО И БЕЗУСЛОВНОГО ПЕРЕХОДА ... 69
1. Цель работы ... 69
2. Теоретический блок ... 69
3. Вопросы для самопроверки ... 75
4. Самостоятельная работа ... 75
Практическое занятие № 2.5. ПРЯМАЯ И КОСВЕННАЯ АДРЕСАЦИЯ ... 75
1. Цели работы ... 75
2. Теоретический блок ... 76
3. Вопросы для самопроверки ... 85
4. Самостоятельная работа ... 86

Практическое занятие № 2.6. ЦИКЛИЧЕСКИЕ ОПЕРАЦИИ. МАССИВЫ. СОРТИРОВКА МАССИВА 86
1. Цели работы 86
2. Теоретический блок 86
3. Вопросы для самопроверки 95
4. Самостоятельная работа 96
Практическое занятие № 2.7. СТЕК 96
1. Цель работы 96
2. Теоретический блок 96
3. Вопросы для самопроверки 102
4. Самостоятельная работа 102
Практическое занятие № 2.8. ПРЕРЫВАНИЯ 102
1. Цель работы 102
2. Теоретический блок 102
3. Вопросы для самопроверки 110
4. Самостоятельная работа 111
Практическое занятие № 2.9. ПРОЦЕДУРЫ 111
1. Цели работы 111
2. Теоретический блок 111
3. Вопросы для самопроверки 122
4. Самостоятельная работа 122
Практическое занятие № 2.10. МАКРОСЫ 123
1. Цели работы 123
2. Теоретический блок 123
3. Вопросы для самопроверки 126
4. Самостоятельная работа 126

Часть 3. MACRO ASSEMBLER: IA-32. ТЕХНОЛОГИЯ WINDOWS API 127
Практическое занятие № 3.1. АРХИТЕКТУРА IA-32. ПЕРЕХОД НА 32-БИТНОЕ ПРОГРАММИРОВАНИЕ 127
1. Цели работы 127
2. Теоретический блок 127
3. Вопросы для самопроверки 131
Практическое занятие № 3.2. ТЕХНОЛОГИЯ WINDOWS API 132
1. Цели работы 132
2. Теоретический блок 132
3. Вопросы для самопроверки 139
4. Самостоятельная работа 139

Практическое занятие № 3.3. НЕПОСРЕДСТВЕННАЯ РАБОТА С ФУНКЦИЯМИ WINDOWS API 140
1. Цели работы 140
2. Теоретический блок 140
3. Вопросы для самопроверки 150
4. Самостоятельная работа 150
Практическое занятие № 3.4. БИБЛИОТЕКА MASM32 150
1. Цель работы 150
2. Теоретический блок 150
3. Вопросы для самопроверки 153
4. Самостоятельная работа 154
Практическое занятие № 3.5. БИБЛИОТЕКА МАКРОСОВ MASM32 154
1. Цель работы 154
2. Теоретический блок 154
3. Вопросы для самопроверки 162
4. Самостоятельная работа 163
Практическое занятие № 3.6. СОЗДАНИЕ ДИНАМИЧЕСКИХ БИБЛИОТЕК 163
1. Цель работы 163
2. Теоретический блок 163
3. Вопросы для самопроверки 171
4. Самостоятельная работа 172
Практическое занятие № 3.7. СТРУКТУРЫ И ОБЪЕДИНЕНИЯ 172
1. Цель работы 172
2. Теоретический блок 172
3. Вопросы для самопроверки 178
4. Самостоятельная работа 178

ЗАКЛЮЧЕНИЕ 179

ПРИЛОЖЕНИЯ 180

БИБЛИОГРАФИЧЕСКИЙ СПИСОК 190

ВВЕДЕНИЕ

Качественная подготовка программиста подразумевает не только владение современными языками программирования, но и понимание механизмов работы компьютера. Высокоуровневые языки программирования обладают большими возможностями, поскольку многолетний опыт разработчиков позволил отобрать наиболее эффективные алгоритмы решения часто возникающих задач. Основная цель – упростить разработку больших проектов.

Однако универсализация обычно скрывает от программиста доступ к «сердцу» компьютера – процессору. С одной стороны, таким образом достигается безопасность кода, однако с другой – приходится жертвовать производительностью программы (которая может быть критически важной в контексте решаемой задачи).

Именно поэтому обучение специалистов в области информатики должно включать курс по архитектуре компьютера, механизмов его работы и программирования. Серьезную роль в формировании компетентности в указанной области играет язык ассемблера, которому посвящен данный практикум.

Ассемблер относят к низкоуровневому языку программирования. Одна из наиболее сильных его сторон – ориентированность на работу конкретного процессора, что позволяет писать максимально возможный по эффективности код в рамках искомой задачи. Помимо этого язык ассемблера лежит в основе современных языков высокого уровня, и изучение его возможностей расширяет понимание принципов их функционирования в целом.

Практикум предназначен для проведения лекционных и лабораторно-практических занятий по дисциплине «Архитектура вычислительных систем и компьютерных сетей», различных спецкурсов по программированию и ориентирован на студентов, не владеющих основами ассемблера, а также на учащихся, имеющих некоторый опыт работы с языком на уровне ассемблерных вставок (например, язык Turbo Pacal).

В книге рассматриваются важнейшие вопросы архитектуры процессоров Intel. В качестве ассемблера выбран программный пакет компании Microsoft – Macro Assembler. Выбор оправдан популярностью этого ассемблера и тем фактом, что после изучения его основ читателю будет весьма просто освоить особенности синтаксиса и функционирования других ассемблеров.

Цель практикума – сформировать базовые и специфические знания ассемблерных инструкций, затрагивающие программирование вплоть до уровня эксперта.

Материал книги разделен на несколько частей. В первой части затронуты вопросы возможностей ассемблера и архитектуры процессоров семейства Intel x86.

Вторая часть посвящена основам Macro Assembler и классической 16-битной архитектуре. Здесь весьма подробно изложены основные инструкции языка, приведены конкретные примеры их использования.

Третья часть описывает особенности программирования в 32-битных архитектурах с использованием Windows API. Этот раздел полезен учащимся, планирующим изучать С++ или владеющим основами этого языка.

Курс выстраивается таким образом, чтобы теоретический материал активно сочетался с практическим. Для полноценного и более эффективного изучения дисциплины в конце каждой темы приведен блок заданий и вопросов для самостоятельной работы. При необходимости их можно выносить на коллективное обсуждение.

1. Цели освоения дисциплины

Основной целью лабораторно-практического курса «Архитектура компьютера: язык программирования ассемблер» является формирование базовых и специфических знаний, умений и навыков по низкоуровневому программированию на ассемблере.

Помимо основной цели преследуется ряд дополнительных:

- расширить понимание основ работы процессора;
- овладеть различными приемами оптимизации программ;
- развить навыки анализа устройства высокоуровневых языков программирования с точки зрения ассемблера.

2. Место дисциплины в структуре основной образовательной программы ВПО

Раздел образовательной программы: Б.3. Профессиональный цикл. Базовая часть.

Для изучения курса необходимы начальные знания по следующим дисциплинам:

- информатика;
- информатика и информационные технологии;
- программирование;
- математическая логика.

Для того чтобы приступить к изучению курса, студент должен **знать:**

- основы систем счисления и двоичной арифметики в частности;
- основы алгебры логики;
- основы теории алгоритмов;
- один из процедурных или объектно-ориентированных языков программирования на уровне консольных приложений;

уметь:

- программировать на одном из процедурных или объектно-ориентированных языков программирования;
- строить математическую и информационную модель процесса.

3. Компетенции обучающегося, формируемые в результате освоения дисциплины (модуля)

В результате освоения данной дисциплины формируются следующие компетенции:

ОК-5 – способность применять знания на практике;

ОК-9 – умение находить, анализировать и контекстно обрабатывать научно-техническую информацию;

ОК-14 – способность к анализу и синтезу;

ПК-7 – умение грамотно пользоваться языком предметной области;

ПК-11 – самостоятельное построение алгоритма и его анализ;

ПК-12 – понимание того, что фундаментальное математическое знание является основой компьютерных наук;

ПК-17 – умение извлекать полезную научно-техническую информацию из электронных библиотек, реферативных журналов, сети Интернет;

ПК-19 – знание математических основ информатики как науки.

В результате освоения дисциплины учащийся должен
знать:

- основные механизмы низкоуровневой разработки;
- основные инструкции и конструкции Macro Assembler;
- механизмы использования регистров процессора и оперативной памяти;
- принципы и различия в типах адресации;
- основы технологии Windows API;

уметь:

- разрабатывать приложения для 16-битного и 32-битного режимов адресации на языке Macro Assembler;
- проводить отладку приложений на уровне ассемблерного кода;
- оптимизировать приложение по скорости работы и занимаемой памяти.

Часть 1. АРХИТЕКТУРА ПРОЦЕССОРОВ INTEL x86

Практическое занятие № 1.1
ЯЗЫК АССЕМБЛЕРА. ОБЗОР АССЕМБЛЕРОВ

1. ЦЕЛИ РАБОТЫ

- Знакомство с понятием языка ассемблера и его возможностями.
- Обзор ассемблеров.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Понятие ассемблера

Ассемблер *(от англ. Assembler – сборщик) – компилятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.*

Как и сам язык, ассемблеры, как правило, специфичны для конкретной архитектуры, операционной системы и варианта синтаксиса языка. Существуют также мультиплатформенные или ограниченно-универсальные ассемблеры. Среди последних можно также выделить группу кросс-ассемблеров, способных собирать машинный код и исполняемые модули (файлы) для других архитектур и операционных систем. Часто ассемблером называют не сам компилятор, а язык программирования.

Ассемблер – язык программирования низкого уровня, представляющий собой формат записи машинных команд, удобный для восприятия человеком. Команды языка соответствуют командам процессора и представляют собой удобную символьную форму записи (мнемокод) команд и их аргументов.

Помимо команд в ассемблере определены собственные директивы.

Директивы *– параметры (ключевые слова) в тексте программы на языке ассемблера, влияющие на процесс ассемблирования или свойства выходного файла.*

Директивы позволяют включать в программу блоки данных (описанные явно или считанные из файла); повторять определённый фрагмент указанное число раз; компилировать фрагмент по условию; задавать адрес исполнения фрагмента, менять значения меток в процессе компиляции; использовать макроопределения с параметрами и

другое, т. е. в целом упрощают работу программиста и улучшают читабельность кода. Как правило, каждая модель процессора имеет свой набор команд и соответствующий ему язык (диалект) ассемблера.

Достоинства ассемблера

- Доступ к регистрам процессора. Регистры представляют собой особые участки памяти процессора, превосходящие по скорости доступа оперативную память.
- Минимальная избыточность кода (использование меньшего количества команд и обращений в память). Как следствие – бóльшая скорость и меньший размер программы.
- Возможность метапрограммирования.
- Достижение максимальной совместимости для требуемой платформы.
- Непосредственный доступ к аппаратуре: портам ввода-вывода, особым регистрам процессора.

Недостатки ассемблера

- Зачастую большой объем кода (особенно для сложных и оконных приложений).
- Плохая читабельность кода, трудность поддержки (отладка, добавление возможностей).
- Трудность реализации различных парадигм программирования, сложность совместной разработки.
- Малое количество доступных библиотек.
- Ассемблер как таковой некросплатформенен.

В связи с вышесказанным можно сделать вывод, что ассемблер, как правило, имеет преимущество при решении узкоспециализированных задач, требующих жесткой экономии ресурса, в то время как языки высокого уровня удобны для больших проектов (задача предельной оптимизации не стоит).

Область применения и роль изучения

- На ассемблере пишутся драйверы и ядра операционных систем.
- Его используют для «прошивки» BIOS.
- На ассемблере пишут компиляторы и интерпретаторы языков высокого уровня.
- Он эффективен для написания вирусов и используется при взломе программ.

Изучение ассемблера полезно как начинающему, так и опытному программисту.

- Ассемблер способствует развитию навыков эффективного программирования, изучению приемов оптимизации и отладки кода.
- Позволяет лучше понять устройство процессора и архитектуры ПК в целом.
- Это язык процессоров и микроконтроллеров.
- Программирование на ассемблере формирует особую культуру мышления программиста.

Рекомендации

Изучение любого нового языка программирования требует сосредоточенности, терпения и постоянной практики. Ассемблер – не исключение.

- Эффективное изучение требует длительной подготовки, в частности, чтения дополнительной литературы.
- Ассемблер – низкоуровневый язык, требующий от программиста постоянного контроля действий.
- Отладчик – верный друг программиста на ассемблере. Он позволяет отслеживать ошибки, которые не всегда очевидны.
- Постоянная практика. Сложные на первый взгляд манипуляции со временем переходят в опыт и выполняются автоматически.

Обзор ассемблеров

Наличие различных ассемблерных языков говорит об отсутствии единого универсального ассемблера, способного одинаково эффективно решать различные задачи.

Выделим наиболее распространенные ассемблерные языки, предназначенные для различных процессоров и микроконтроллеров.

Turbo Assembler (TASM) – продукт компании Borland, предназначенный для разработки программ на языке ассемблера для архитектуры x86. Кроме того, TASM способен работать совместно с трансляторами с языков высокого уровня фирмы Borland, такими как Turbo C и Turbo Pascal.

TASM до сих пор используется для обучения программированию на ассемблере под MS-DOS. Многие находят его очень удобным и продолжают использовать, расширяя набором дополнительных макросов.

TASM способен работать в режиме совместимости с Macro Asembler (MASM). Кроме того, TASM имеет собственный режим IDEAL, улучшающий синтаксис языка и расширяющий его функциональные возможности.

Как и прочие программные пакеты серии Turbo, TASM официально больше не поддерживается, поэтому считается «умирающим» языком. Однако последние версии (TASM 5.0) позволяют писать 32-битные приложения.

Macro Assembler (MASM) – ассемблер для процессоров семейства x86. Первоначально был произведён компанией Microsoft для написания программ в операционной системе MS-DOS и был в течение некоторого времени самым популярным ассемблером, доступным для неё. Это поддерживало широкое разнообразие макросов и структурированность программных идиом, включая конструкции высокого уровня для повторов, вызовов процедур и чередований (поэтому MASM – ассемблер высокого уровня). Позднее была добавлена возможность написания программ для Windows.

В связи с широкой распространённостью MASM посвящено большое количество учебной литературы (в том числе и русскоязычной) и готовых библиотек. Это позволяет называть MASM самым популярным ассемблером из доступных.

Flat assembler (FASM) – свободно распространяемый многопроходной ассемблер, написанный Томашем Грыштаром (польск. Tomasz Grysztar). Fasm написан на самом себе, обладает небольшими размерами и высокой скоростью компиляции, мощным макросинтаксисом, позволяющим автоматизировать множество рутинных задач. Поддерживаются как объектные форматы, так и форматы исполняемых файлов.

FASM – многоплатформенный ассемблер: на нем можно программировать как в Windows, так и в Unix-системах.

NASM (Netwide Assembler) – свободный ассемблер для архитектуры Intel x86. Используется для написания 16-, 32- и 64-битных программ.

NASM компилирует программы под различные операционные системы в пределах x86-совместимых процессоров. Находясь в одной операционной системе, можно беспрепятственно откомпилировать исполняемый файл для другой.

YASM – название имеет несколько возможных истолкований: Yes, it's an ASseMbler, Yet Another aSseMbler, Your fAvorite aSseMbler и др. YASM – попытка полностью переписать ассемблер NASM. В настоящее время развивается Питером Джонсоном и Майклом Ерманом.

YASM предлагает поддержку x86-64. Кроме Intel-синтаксиса, применяемого в NASM, YASM также поддерживает AT&T-синтаксис, распространённый в Unix. YASM построен «модульно», что позволяет легко добавлять новые формы синтаксиса, препроцессоры и т. п.

GNU Assembler, или **GAS** – ассемблер проекта GNU; используется компилятором GCC. Входит в пакет GNU Binutils. Кроссплатформенная программа запускается и компилирует код для многочисленных процессорных архитектур. Распространяется на условиях свободной лицензии GPL 3.

GAS был разработан для поддержки компиляторов Unix, он использует стандартный синтаксис AT&T, который несколько отличается от большинства ассемблеров, основанных на синтаксисе Intel.

RosAsm – 32-битовый Win32 x86 ассемблер. Согласно своему имени, ассемблер поддерживает операционную систему ReactOS, хотя проекты RosAsm и ReactOS независимы. RosAsm – среда разработки с полной интеграцией ассемблера, встроенного линкера, редактора ресурсов, отладчика и дизассемблера. Синтаксис сделан как продолжение NASM'а.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что называют ассемблером?
2. Чем команды ассемблера отличаются от директив?
3. Каковы основные преимущества и недостатки ассемблерного кода?
4. Чем объясняется отсутствие единого ассемблерного языка?
5. Чем полезно изучение ассемблера начинающему и опытному программисту?

Практическое занятие № 1.2

ПРЕДСТАВЛЕНИЕ ЧИСЕЛ В КОМПЬЮТЕРЕ. МОДЕЛЬ ПАМЯТИ

1. ЦЕЛИ РАБОТЫ

- Изучение алгоритма представления целых чисел со знаком и без знака.
- Построение формальной модели оперативной памяти.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Системы счисления

Логические схемы компьютера условно могут различать два состояния: включено (логическая единица) и выключено (логический нуль). Поэтому любая информация в компьютере представляется в форме двоичного кода (т. е. чисел), а вычислительные операции строятся на основе двоичной алгебры.

Двоичная система счисления является естественной для компьютера, но малоудобной для программиста. Основные операции мы осуществляем через десятичную систему счисления. Однако она не позволяет анализировать представление информации естественным образом; требуется сначала перевести число в двоичную форму.

Таким образом, двоичная форма числа естественна, однако запись даже небольших чисел в ней получается весьма громоздкой. Поэтому гораздо чаще операции производят в шестнадцатеричной системе счисления.

Вне зависимости от используемой системы счисления ассемблер переводит число в двоичную форму.

Перевод целых неотрицательных чисел

Ассемблер способен работать с двоичной, восьмеричной, десятичной и шестнадцатеричной системами счисления. Чаще всего в ассемблерных программах используется шестнадцатеричная форма числа, поскольку с ней удобно оперировать; отладчики выдают информацию в этой системе счисления.

Перевод из десятичной в двоичную систему

Для перевода в двоичную систему счисления требуется делить число на 2, пока неполное частное не меньше двух. Остатки от деления в обратном порядке задают цифры двоичного числа.

Например, разложение для 2015:

Неполное частное	2015	1007	503	251	125	62	32	15	7	3	1
Остаток	1	1	1	1	1	0	1	1	1	1	

Таким образом, $2015_{10} = 11111011111_2$.

Перевод из десятичной в шестнадцатеричную систему

Для перевода в шестнадцатеричную систему счисления требуется делить число на 16, пока неполное частное не меньше 16. Остатки от деления в обратном порядке задают цифры шестнадцатеричного числа.

Например, разложение для 2015:

Неполное частное	2015	125	7
Остаток	15 (F)	13 (D)	

Таким образом, $2015_{10} = 7\text{DF}_{16}$.

Перевод из двоичной системы в шестнадцатеричную и обратно

Для перевода двоичного числа в шестнадцатеричное его разбивают на секции по четыре разряда, начиная от младших разрядов. Каждая двоичная секция соответствует одной шестнадцатеричной цифре.

Например:

$$11111011111_2 = 111.1101.1111_2 = 7\text{DF}_{16}.$$

Для перевода шестнадцатеричного числа в двоичное каждую цифру числа заменяют четырьмя двоичными цифрами.

Например:

$$7\text{DF}_{16} = 0111.1101.1111_2 = 11111011111_2.$$

Перевод в десятичную систему

Для перевода двоичного числа в десятичное требуется разложить его в линейную комбинацию по степеням двойки.

Например:

$$11111011111_2 = 1 \cdot 2^{10} + 1 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + \\ +0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 2015_{10}.$$

Для перевода шестнадцатеричного числа в десятичное требуется разложить его в линейную комбинацию по степеням числа 16.

Например:

$$7DF_{16} = 7 \cdot 16^2 + 13 \cdot 16^1 + 15 \cdot 16^0 = 2015_{10}.$$

Формальная модель оперативной памяти

Несмотря на историческую преемственность архитектуры процессоров Intel, модель адресации памяти претерпела серьезные изменения. Однако это не мешает нам построить формальную модель, которая в целом будет корректно описывать реально существующие.

Измерение количества информации

Любой разряд двоичного числа может принимать значение 0 либо 1. Ячейка в один разряд называется битом и является наименьшей единицей измерения количества информации. 1 бит способен закодировать два числа, два бита – 4 числа, три бита – 8 чисел и т. д.

Общий закон, очевидно, задается по правилу[1]

$$I = 2^i,$$

где I – количество двоичных чисел, которые можно закодировать; i – количество бит.

Бит – очень маленькая ячейка памяти. Поэтому операции осуществляются над ячейками, кратными 8 битам, именуемыми байтами. Большие объемы памяти обычно переводятся в более высокие порядки единиц измерения.

Единица измерения	Обозначение	Степень двойки	Число байт
Килобайт	Кб	10	1024
Мегабайт	Мб	20	1024^2
Гигабайт	Гб	30	1024^3
Терабайт	Тб	40	1024^4

Адресация оперативной памяти

Для обращения к ячейке оперативной памяти процессор должен знать ее адрес. Схематично устройство оперативной памяти можно представить в виде ленты подряд идущих байтов.

[1] Читателю этот закон должен быть знаком по формуле Хартли.

Адрес	...	2012	2013	2014	2015	2016	...
Значение	...	56	2F	00	FE	A8	...

Однако для однозначных операций с ячейками оперативной памяти кроме адреса необходимо знать и размер ячейки – количество однобайтовых ячеек, которое занимает значение. Если речь идет о переменных, то размер определяется типом переменной.

Перевод целых чисел со знаком

Пусть формально имеется ячейка размером в k бит, тогда можно составить 2^k различных чисел. Если числа неотрицательные, то они входят в диапазон $0..2^k - 1$.

Для возможности работы с отрицательными числами половину диапазона чисел отводят на отрицательные, а другую на неотрицательные, т. е. в общем случае число со знаком лежит в диапазоне $-2^{k-1}..2^{k-1} - 1$. Отрицательные числа представляются в дополнительном коде.

Дополнительный код числа можно задать следующей функцией:

$$\text{доп}(x) = \begin{cases} x, \text{ если } x \geq 0; \\ 2^k - |x|, \text{ если } x < 0. \end{cases}$$

Ограничимся ячейкой размером в байт ($k = 8$):

$\text{доп}(1) = 1 = 00000001$	$\text{доп}(-1) = 256 - 1 = 11111111$
$\text{доп}(2) = 2 = 00000010$	$\text{доп}(-2) = 256 - 2 = 11111110$
$\text{доп}(3) = 3 = 00000011$	$\text{доп}(-3) = 256 - 3 = 11111011$
...	...
$\text{доп}(126) = 126 = 01111110$	$\text{доп}(-126) = 256 - 126 = 10000010$
$\text{доп}(127) = 127 = 01111111$	$\text{доп}(-127) = 256 - 127 = 10000001$
$\text{доп}(0) = 0 = 00000000$	$\text{доп}(-128) = 256 - 128 = 10000000$

Очевидно, что старший бит числа можно рассматривать как флажок знака: у отрицательных чисел он равен 1.

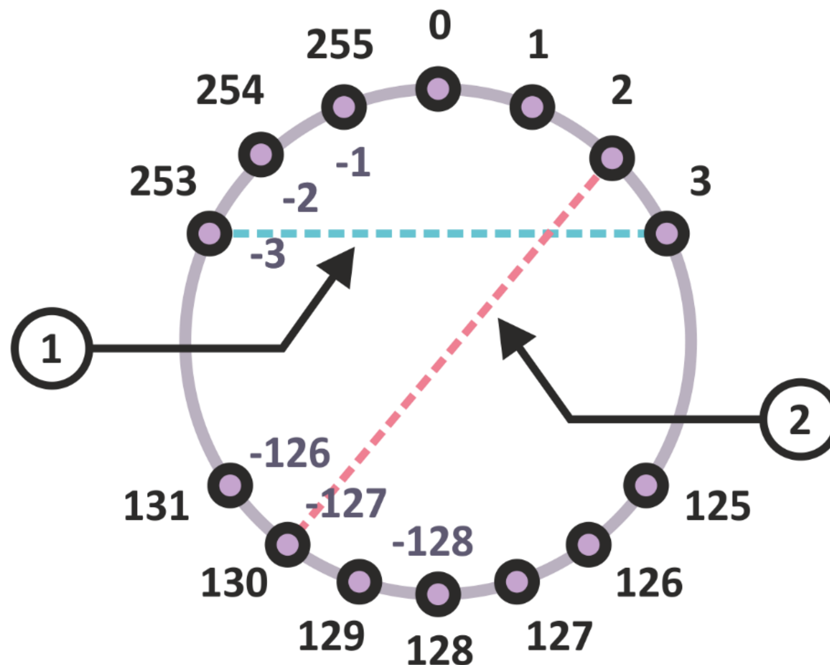
Дополнительный код упрощает разработку логических схем. Так, операция разности замещается сложением. В частности, сумма двух противоположных чисел должна быть равной нулю.

Действительно, сложим, например, 3 и –3:

$3 + (-3) = 00000011 + 1111101 = 1.00000000$ (занимает 9 бит).

Поскольку мы ограничены ячейкой размером в байт, 1 из старшего разряда игнорируется, а в качестве результата берется 0.

Логику построения дополнительного кода хорошо иллюстрирует следующая модель. Представим, что числа от 0 до 255 расположены по окружности в порядке возрастания. Если в противоположную сторону отметить отрицательные числа, то заметим, что соответствующие им числа – дополнительный код:



Очевидно следующее:

- сумма противоположных чисел всегда равна 256, что с учетом «заворота» дает нуль (указатель 1);
- диаметрально противоположные числа в двоичном представлении отличаются лишь знаковым битом (указатель 2).

Дополнительный код можно получить другим способом, если число уже в двоичной или шестнадцатеричной форме.

Алгоритм получения дополнительного кода с помощью инверсии

Пусть дано отрицательное число x.

1. Переведем модуль x в двоичную форму.

2. Инвертируем биты числа (нули заменяем единицами, единицы – нулями).

3. К полученному числу добавим 1.

Например:
1. $-10 \rightarrow 10 = 00001010$.
2. $00001010 \rightarrow 11110101$.
3. $11110101 \rightarrow 11110101 + 1 = 11110110$.

Пусть дано отрицательное число x.
1. Переведем модуль x в шестнадцатиричную форму.
2. Инвертируем цифры числа (каждую цифру вычитаем из 15).
3. К полученному числу добавим 1.

Например:
1. $-125 \rightarrow 125 = 7\mathrm{D}$.
2. $7D \rightarrow [15 - 7][15 - 13] = 82$.
3. $82 \rightarrow 82 + 1 = 83$.

Различие знаковых и беззнаковых чисел

Предположим, в некотором байте памяти хранится число FE. Как понять, что это число 254 либо –2?

Для случайно взятой ячейки памяти нельзя определить однозначно, какое из чисел в ней записано.

Однозначность возможна только в случае, если нам известно, что заданный участок памяти является переменной (данными) определенного типа.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Какие системы счисления используются в ассемблерном коде?
2. Сколько целых неотрицательных чисел можно закодировать в ячейку размерами в 2 байта, 32 бита? В каком диапазоне можно закодировать числа со знаком, занимающие в памяти 3 байта?
3. Каким образом осуществляется перевод числа из двоичной системы счисления в шестнадцатеричную и обратно?
4. Опишите формальную модель устройства оперативной памяти.
5. Что такое дополнительный код? Почему говорят, что дополнительный код обладает свойством инверсии относительно сложения?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Переведите числа в указанную систему счисления:
   - $509_{10} \rightarrow ***_2$ и обратно;
   - $187_{10} \rightarrow ***_{16}$ и обратно;

- $10111010_2 \rightarrow ***_{16}$ и обратно;
- $9FA_{16} \rightarrow ***_2$ и обратно.

2. Укажите минимальное число бит и байт, требуемое для хранения указанных чисел:
   - 34; 65; 309;
   - 101_2; 1100001_2; 11100011110110011_2;
   - 0_{16}; 56_{16}; $1F67A56_{16}$.
3. Получите дополнительный код указанных чисел (в двоичной и шестнадцатеричной формах), если операции замкнуты относительно двухбайтовой ячейки:
   - –6; -1023.
   - -101001_2; -101111100111_2;
   - $-6E_{16}$; $1F09_{16}$.
4. Покажите, что дополнительный код нуля является нулем.
5. Найдите дополнительный код числа -128 (в рамках одного байта). Какой вывод можно сделать?
6. Продемонстрируйте корректность сложения любых чисел на основе дополнительного кода:
   - двух положительных;
   - положительного и отрицательного (отрицательное меньше по модулю);
   - положительного и отрицательного (отрицательное больше по модулю);
   - двух отрицательных.

Практическое занятие № 1.3

ПРОЦЕССОРЫ СЕМЕЙСТВА Intel x86. РЕЖИМЫ АДРЕСАЦИИ

1. ЦЕЛИ РАБОТЫ

- Кратко ознакомиться с историей развития процессоров Intel x86.
- Изучить режимы работы процессора и отличия между ними.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Здесь и далее рассматриваются процессоры семейства Intel x86. Выбор этой линии обосновывается популярностью процессоров, а также наследованием концепции архитектуры на протяжении длительного времени.

Intel 8086 – первый 16-битный микропроцессор компании Intel, выпущенный 8 июня 1978 года. Процессор содержал набор команд, которые поддерживаются и в современных процессорах; именно от этого процессора берёт своё начало известная сегодня архитектура x86.

Регистр процессора – блок ячеек сверхбыстрой оперативной памяти внутри процессора. Регистры используются самим процессором и в основном недоступны программисту. Воспользоваться регистрами непосредственно можно только через язык ассемблера.

Характерной особенностью процессора 8086 являлась специфическая схема адресации (деление памяти на сегменты). Размер регистров составлял 16 бит (2 байта). Процессор работал исключительно в *реальном режиме* адресации.

Intel 80286 (i286) – 16-битный x86-совместимый микропроцессор второго поколения компании Intel, выпущенный 1 февраля 1982 года. Являлся усовершенствованным вариантом процессора Intel 8086, но обладал в 3 – 6 раз большей производительностью. Главное нововведение: помимо реального режима адресации появляется *защищенный*, что позволило исключить недостатки реальной адресации.

Intel 80386 (i386, 386) – 32-битный x86-совместимый процессор третьего поколения фирмы Intel, выпущенный 17 октября 1985 года. Данный процессор был первым 32-разрядным процессором для IBM PC-совместимых ПК. Применялся преимущественно в настольных и портативных ПК (ноутбуки и лэптопы). В настоящее время в основном используется в контроллерах, а также в бытовой технике.

В отличие от предшественника i286 процессор мог работать в реальном, защищенном и виртуальном режимах адресации. Это позволяло запускать несколько программ, написанных для реального режима, в форме отдельных «виртуальных машин». Иными словами, программы выполнялись в параллельном режиме (что было невозможно в MS-DOS). Существенно расширяются возможности работы с памятью.

В i386 впервые применялись 32-битные регистры. Благодаря ему высокую популярность приобрела операционная система Windows. Процессор i386 положил начало семейству архитектур IA-32 (Intel Architecture, 32-bit).

Реальный режим адресации

Процессор 8086 мог работать только в реальном режиме адресации памяти. Все следующие модели, начиная с процессора 80286, сохранили режим совместимости с 8086.

Наименьшим адресуемым блоком памяти является байт (8 бит). Каждый байт памяти имеет уникальное местоположение – *физический адрес*, по которому в него может записываться и читаться информация. Для получения доступа к ячейке памяти процессору необходимо знать ее физический адрес.

Для доступа к памяти процессор использует *адресную шину*, на которой выставляет адрес требуемой ячейки памяти.

Если адресная шина имеет размер 1 бит, то она может адресовать два байта памяти. Двухбитная адресная шина способна адресовать уже 4 байта памяти. Трехбитная адресная шина адресует 8 байт памяти и т. д. Таким образом, чем больше разрядность у адресной шины, тем больший объем памяти можно адресовать.

Процессор 8086 имел 20-битную адресную шину, что позволяло адресовать $2^{20} = 1048576$ байт памяти (1 Мбайт). Однако процессор 8086 имел 16-битную архитектуру. А с помощью 16 бит можно адресовать только $2^{16} = 65536$ байт памяти (64 Кбайт). Каким же образом процессор 8086 адресовал 1 Мбайт памяти?

Решением стала *сегментная адресация* памяти. С помощью этого метода физический адрес конкретного байта памяти *логически* определяется двумя 16-разрядными значениями. Для того чтобы с помощью 16-разрядных регистров можно было обращаться в любую точку 20-разрядного адресного пространства, введён двухкомпонентный логический адрес из двух 16-разрядных компонент:

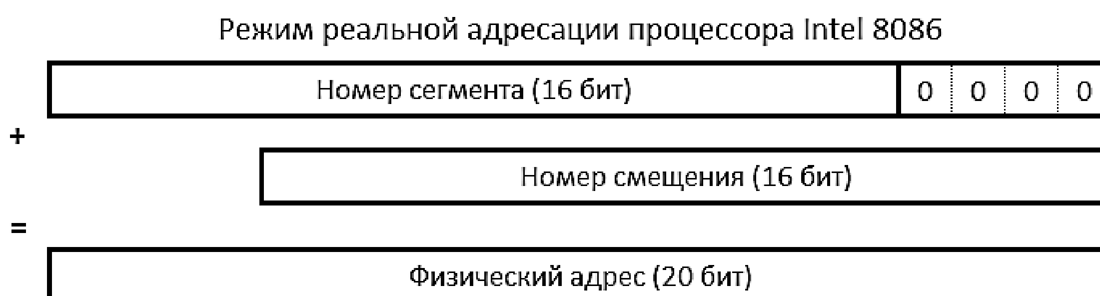
Segment : Offset (Сегмент : Смещение),

где Segment – адрес сегмента, а Offset – смещение от начала этого сегмента. Такую схему можно представить в форме многоэтажной гостиницы: сегменты будут обозначать номер этажа, а смещение – номер комнаты (допустим, что на каждом этаже нумерация начинается заново).

Для определения начала сегментов памяти процессор 8086 использует четыре 16-битных сегментных регистра (CS, DS, SS, ES).

Смещение внутри сегмента выбирается из регистров-указателей SP, BP, SI, DI или регистра IP.

Для получения 20-битного физического адреса процессор размещает на адресной шине значение сегментного регистра и сдвигает его влево на четыре бита, заполняя младшие четыре бита адресной шины нулями (равносильно умножению на десятичное 16 или шестнадцатеричное 10), затем к этому значению прибавляется смещение. Адрес сформирован.



Например, имеем логический адрес 3E76:1A21. Согласно правилу значение сегмента умножаем на 10_{16} или, что то же самое, сдвигаем на 4 бита влево

3E76 · 10 = 3E760.

К полученному 20-разрядному значению прибавляем смещение

3E760 + 1A21 = 40181h.

Полученное число – физический (реальный) адрес.

Получается, что границы сегментов (16-битное значение + 4 нулевых бита) располагаются через каждые 16 байт физических адресов. 4 битами можно адресовать 16 (байт) ячеек памяти. Каждый из этих 16-байтовых фрагментов называется *параграфом*. 16-разрядные сегментные регистры могут адресовать $2^{16} = 65536$ параграфов (границ сегментов). А параграф, как уже говорилось, это 16 байт. 65536 параграфов по 16 байт каждый дают всего 1048576 байт или 1 Мбайт памяти, что и требовалось.

Максимальный размер сегмента определяется теми же 16 битами регистра, в котором хранится смещение. Следовательно, максимальный размер сегмента может быть $2^{16} = 65536$ байт (64 Кб). Минимальный – 16 байт (размер параграфа). Таким образом, сегменты –

это виртуальные умозрительные части с максимальным объемом 64 Кбайт каждая.

Однако такой подход имеет серьезный недостаток.

Рассмотрим, к примеру, логический адрес FFFF:FFFF. По правилу образования физического адреса получим FFFF0 + FFFF = = 10FFEF. Мы вышли за пределы 20-битной адресной шины: она может определить диапазон с адресами от 00000 до FFFFF. При адресации памяти свыше FFFFF происходит «заворот» – старший единичный бит адреса проигнорируется и доступ идет к 64 килобайтам в начальных адресах, т. е. ссылаемся на ячейку с адресом 0FFEF.

Все это может привести к следующим проблемам.

- Нет никаких препятствий для обращения к физически не существующей памяти.
- При обращении к несуществующей памяти результат непредсказуем (все зависит от разработчика материнской платы и другого аппаратного обеспечения компьютера).
- Программа может обращаться к любому сегменту как для считывания, так и для записи данных и команд.

Полезно запомнить некоторые факты о сегментации памяти.

- Сегменты физически не выделены в памяти. Сегменты – это логические сущности, через которые программы просматривают области памяти удобными в 64 Кбайт порциями.
- Размеры сегментов могут изменяться от 16 байт до 64 Кбайт.
- Сегменты в памяти не обязательно располагаются один за другим, они могут перекрываться; поэтому один и тот же физический байт памяти может иметь различные логические адреса, определяемые разными, но при этом эквивалентными парами сегмент – смещение. Например, пары логических адресов 0000:0010 и 0001:0000 указывают на один и тот же физический адрес ячейки памяти – 00010.
- Назначением базовых адресов сегментов занимается операционная система, а внутри каждого сегмента адреса формируются программой.

- Сегментная организация обеспечивает создание позиционно независимых или динамически перемещаемых в памяти программ.

Защищенный режим адресации

В реальном режиме процессор может обращаться только к первому мегабайту памяти, адреса которого находятся в диапазоне от 00000 до FFFFF. При этом процессор работает в однопрограммном режиме (т. е. в заданный момент времени способен выполнять только одну программу). При этом он может в любой момент прервать ее выполнение и переключиться на процедуру обработки запроса (ее называют *прерыванием*), поступившего от одного из периферийных устройств (мышь, клавиатура, др.). Любой программе, которую выполняет в этот момент процессор, разрешен доступ без ограничения к любым областям памяти, находящимся в пределах первого мегабайта: к ОЗУ по чтению и записи, а к ПЗУ только для чтения.

Реальный режим работы процессора использовался в операционной системе MS-DOS, а также в системах Windows 95 и Windows 98 при загрузке в режиме эмуляции MS-DOS.

Однако начиная с процессора 80286 вводится понятие нового режима работы – *защищенного*. Первая полноценная реализация защищенного режима становится возможной с появлением нового семейства процессоров 80386, содержащим уже 32-битные регистры.

В защищенном режиме можно реализовать все возможности процессора, вложенные разработчиками. При работе в защищённом режиме каждой программе может быть выделен блок памяти размером до $2^{32} = 4$ Гбайт, адреса которого в шестнадцатеричном представлении могут меняться от 00000000 до FFFFFFFF. При этом говорят, что программе выделяется *линейное адресное пространство* (flat address space), которое разработчики компилятора Microsoft Assembler назвали линейной моделью памяти (flat model).

С точки зрения программиста линейная модель наиболее проста в использовании, поскольку для хранения адреса любой переменной или команды достаточно одного 32-разрядного целого числа. Во многом это достигается за счет того, что часть работы программиста по реализации встроенных возможностей процессора выполняет операционная система.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что называют регистрами процессора. Какой размер регистров был у процессоров Intel 8086 и Intel 80386?
2. Какие режимы работы процессора существуют? Кратко охарактеризуйте их.
3. Что такое сегментация памяти? Чем отличается физический адрес от логического?
4. В чем преимущество защищенного режима работы процессора перед реальным?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Задан логический адрес 5E20:FF37. Определить соответствующий ему физический адрес. Операции проделать в двоичной и шестнадцатеричной системе счисления.
2. Доказать, что в защищенном режиме можно адресовать 4 Гб ОЗУ.

Практическое занятие № 1.4

РЕГИСТРЫ ПРОЦЕССОРА. ОПЕРАТИВНАЯ ПАМЯТЬ

1. ЦЕЛИ РАБОТЫ

- Изучить систему регистров 16- и 32-битных процессоров.
- Рассмотреть логическую структуру оперативной памяти.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Процессор Intel 8086

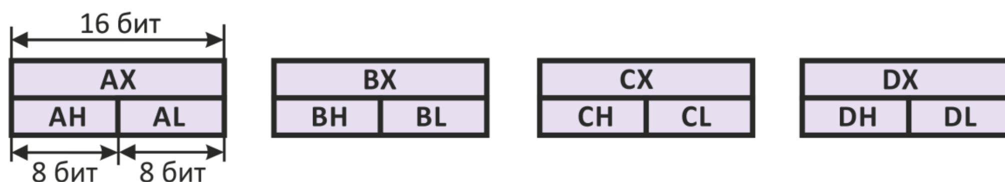
Процессор Intel 8086 имел 14 16-битных регистров, фактически разделённых на несколько функциональных категорий:

- регистры общего назначения (AX, BX, CX, DX);
- сегментные регистры (CS, DS, SS, ES);
- индексные регистры (SI, DI);
- регистры-указатели (BP, SP, IP);
- регистр флагов.

Имена регистров являются также и идентификаторами в ассемблерных программах, по которым к ним обращаются.

Регистры общего назначения

Всего четыре регистра с именами AX, BX, CX и DX. Их можно обрабатывать как единую двухбайтовую ячейку памяти и по отдельности старший и младший байты соответственно



Регистры общего назначения используются в арифметических операциях и пересылке данных. Некоторые команды «привязаны» к конкретным регистрам.

AX – аккумулятор (accumulator). Используется для хранения операндов в командах умножения и деления, ввода-вывода, в некоторых командах обработки строк и других операциях.

BX – регистр базы (base). Предназначен для хранения адреса или части адреса операнда, находящегося в памяти.

CX – счётчик (counter). Содержит количество повтора цикла, сдвигов и строковых операций.

DX – регистр данных (data). Используется для косвенной адресации портов ввода-вывода, а также как «расширитель» в паре с AX.

Сегментные регистры

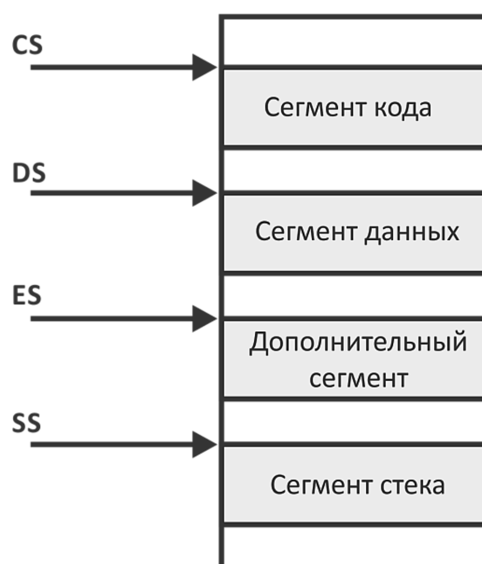
Процессор включает четыре сегментных регистра: CS, DS, SS, ES.

CS (Code Segment) – содержит начальный адрес сегмента кода программы. Этот адрес вместе со значением регистра IP определяет адрес команды для выполнения.

DS (Data Segment) – хранит начальный адрес сегмента данных.

SS (Stack Segment) – содержит начальный адрес сегмента стека.

ES (Extra Segment) – является вспомогательным регистром, используется при некоторых операциях над строками.



Схематично изображено распределение сегментов в 1МБ памяти (порядок сегментов может быть другим, а сегменты – перекрываться).

Индексные регистры

Имеется два индексных регистра: SI и DI.

SI (Source Index) – индекс источника; применяется для некоторых операций над строками (обычно в паре с регистром DS);

DI (Destination Index) – индекс приемника; применяется для строковых операций (обычно в паре с регистром ES).

Оба регистра используются в режимах адресации, операциях сложения и разности.

Регистры-указатели

SP (Stack Pointer) – указывает на вершину стека, позволяет временно хранить адреса и данные;

BP (Base Pointer) – обычно адресует переменные, хранимые в стеке, облегчает доступ к параметрам (данным и адресам), переданным через стек.

IP (Instruction Pointer) – содержит адрес команды, которая должна быть выполнена. Поэтому менять значение регистра не рекомендуется.

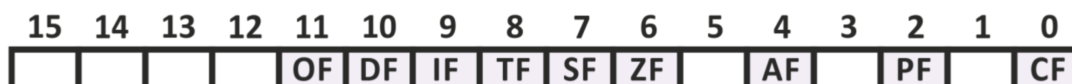
SP и BP обеспечивают доступ к данным в сегменте стека.

Регистр флагов

Регистр флагов хранит состояние процессора.

В дальнейшем мы будем говорить, что бит устанавливается, если он принимает значение 1, и сбрасывается, если 0.

Биты регистра устанавливаются или сбрасываются в зависимости от результата исполнения предыдущей команды. Ряд команд ассемблера считывает данные из этого регистра для выполнения требуемой операции. Биты регистра могут также устанавливаться и сбрасываться специальными командами процессора. Отдельные биты флагов представляют одиночными буквами O, D, I, T, S, Z, A, P, C или двумя буквами OF, DF, IF, TF, SF, ZF, AF, PF и CF.



CF (Carry Flag) – флаг переноса при арифметических операциях. Содержит перенос из старшего бита после арифметических операций, а также последний бит при сдвигах или циклических сдвигах.

PF (Parity Flag) – флаг четности результата. Устанавливается в 1, если в результате операции получено число с четным количеством 1, иначе 0.

AF (Auxiliary Flag) – флаг дополнительного переноса. Устанавливается при возникновении переноса или заёма из 4-го разряда в 3-й. Сбрасывается при отсутствии такового. Используется командами десятичной коррекции.

ZF (Zero Flag) – флаг нулевого результата. Показывает результат арифметических операций и операций сравнения: устанавливается в 1, если результат получился нулевой.

SF (Sign Flag) – флаг знака. Совпадает со старшим битом результата: 0, если без знака, и 1 в противном случае.

TF (Trap Flag) – флаг пошагового режима (используется при отладке).

IF (Interrupt-enable Flag) – флаг аппаратных прерываний. Если равен нулю, то прерывания запрещены.

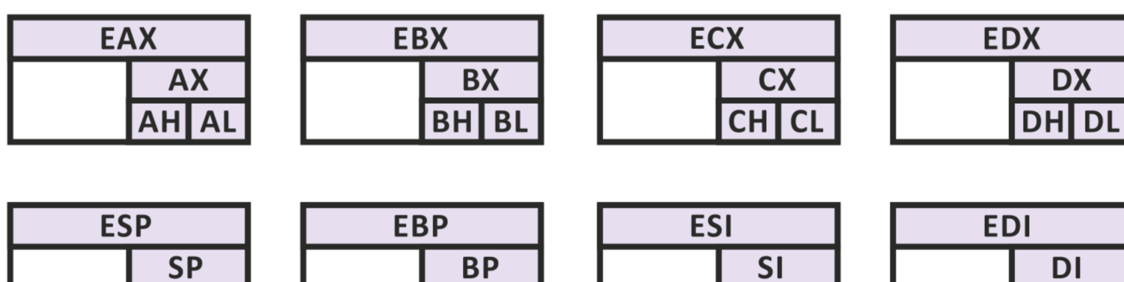
DF (Direction Flag) – флаг направления при строковых операциях. Обозначает левое или правое направление пересылки или сравнения строковых данных.

OF (Overflow Flag) – флаг переполнения. Указывает на переполнение старшего бита при арифметических командах.

Регистры процессора i386. Архитектура IA-32

Как было сказано ранее, начиная с процессора 80386 внедряются 32-битные регистры. По существу это расширение архитектуры с 16 бит на 32. Все регистры (за исключением сегментных) стали 32-битными, получив в названии префикс «E» (EAX, EBX, EIP, EFLAGS и т. п.), с сохранением полного набора команд для работы с ними. В частности, регистр флагов получил новые флаги для управления многозадачностью.

Общую схему доступных регистров можно представить следующим образом:



Регистр EIP полностью заменил 16-битного предшественника IP.

Регистр флагов FLAGS расширением до EFLAGS пополнился дополнительными флагами в старшей части.

Сопроцессор

Семейство процессоров IA-32 содержит *модуль операций с плавающей точкой (FPU)*. В процессорах Intel 386 этот блок присутствовал в качестве сопроцессора, обозначаемого Intel 387.

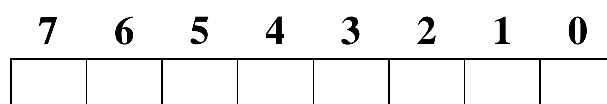
Сопроцессор позволяет производить обычные арифметические операции, а также операции с трансцендентными выражениями.

В модуле FPU можно выделить три группы регистров.

- Стек процессора: 8 регистров R0, R1 … R7 размером 80 бит каждый.
- Служебные регистры:
  - регистр состояния процессора SWR (Status Word Register) хранит информацию о текущем состоянии сопроцессора. Размерность 16 бит;
  - управляющий регистр сопроцессора CWR (Control Word Register) – управление режимами работы сопроцессора. Размерность 16 бит;
  - регистр слова тегов TWR (Tags Word Register) – контроль за регистрами R0…R7. Размерность 16 бит.
- Регистры указателей:
  - указатель данных DPR (Data Point Register). Размерность 48 бит;
  - указатель команд IPR (Instruction Point Register). Размерность 48 бит.

Оперативная память

Оперативная память состоит из ячеек, каждая из которых имеет 8 разрядов (бит). Такие ячейки называют **байтами**. Разряды расположены по старшинству: от младшего 0-го до старшего 7-го.



Байт является наименьшей адресуемой ячейкой памяти.

Существуют и более крупные блоки: слова, двойные слова и т. д.

Слово – два соседних байта (индексация битов от 0 до 15).

Двойное слово – два соседних слова (индексация битов от 0 до 31).

Для слова, двойного слова и любой ячейки памяти более одного байта в качестве адреса берут адрес младшего байта.

Ячейку памяти однозначно определяют две компоненты:

- её адрес (адрес младшего байта);
- размер ячейки (тип).

Байты, слова, двойные слова (и др.) можно обрабатывать целиком или по отдельности (каждый или несколько байт).

Ранее мы построили формальную модель адресации памяти. Ее можно использовать для интерпретации как реального, так и защищенного режима адресации. А именно, защищенный режим весьма полно описывается моделью ленты проиндексированных по возрастанию байт (индекс – это адрес). В реальном режиме память сегментируется, однако программисту редко приходится вручную устанавливать сегментные адреса, так как это делает система. Поэтому (не нарушая общности) считаем сегмент фиксированным, а в качестве адреса берем смещение, т. е. внутри сегмента вновь работает модель ленты.

В дальнейшем многие примеры будут содержать числовые константы не в десятичной системе счисления, а в шестнадцатеричной и реже двоичной. Работа в указанных системах естественным образом связана с битами/байтами.

Например, двоичный байт

0	0	1	1	1	1	0	1

легко конвертировать в шестнадцатеричный

3	D

и обратно.

Ранее было указано, что адрес слов, двойных слов (и т. д.) определяется младшим байтом ячейки памяти. Такая адресация неслучайна, а следует из особенности индексации байтов в архитектуре x86:

Слова, двойные слова (и более крупные блоки памяти) хранятся в «перевернутом» виде, т. е. младшие байты смежны со старшими слева направо.

Например, $245_{10} = 1111\,0101_2$ хранится в слове следующим образом:

1	1	1	1	0	1	0	1	0	0	0	0	0	0	0	0
1-й байт								2-й байт							

а $10\,025\,863_{10} = 98\,FB\,87_{16}$ в двойном слове запишется так:

8	7	F	B	9	8	0	0
1-й байт		2-й байт		3-й байт		4-й байт	

Например, по адресу 0019 хранится байт со значением 12, а по адресу 001A – двойное слово со значением F54:

0017	0018	0019	001A	001B	001C	001D	001E	001F
...	...	12	54	0F	00	00	...	...

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Перечислите регистры Intel 8086 и их функциональное предназначение. Чем они отличаются от регистров Intel 80386?
2. Для чего предназначен регистр флагов?
3. Какие функции выполняет сопроцессор?
4. Что берется в качестве адреса ячейки памяти, большей одного байта?
5. В каком порядке записываются байты числа в оперативную память согласно архитектуре Intel x86?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. В регистре BX хранится число $4BB5_{16}$. Чему будут равны старшая и младшая части регистра?
2. Дана схема участка оперативной памяти:

...	2CDB	2CDC	2CDD	2CDE	2CDF	2CE0	2CE1	2CE2	...
...	56	0F	00	01	00	00	45	FF	...

Укажите:

- адреса байтов, содержащих значение 0;
- адреса слов, содержащих значение 0;
- адрес и размер закрашенной ячейки.

3. Схематично изобразите распределение в памяти слова $3F_{16}$, байта 11001_2 и двойного слова $AF00345_{16}$ (считать, что числа записываются подряд).

Практическое занятие № 1.5
СБОРКА АССЕМБЛЕРНОЙ ПРОГРАММЫ

1. ЦЕЛИ РАБОТЫ

- Ознакомиться с понятием среды разработки и ее функциями.
- Изучить основные компоненты ассемблерной программы.
- Научиться компилировать программу с командной строки.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Понятие среды разработки

Разработка программы – процесс, состоящий из нескольких этапов. Как правило, основная работа программиста связана с реализацией программного кода. Остальные этапы осуществляются неявно средой разработки.

Интегрированная среда разработки *(далее* ***IDE****) – система программных средств, используемая программистами для разработки программного обеспечения.*

Основная задача любой IDE – упрощение работы программиста. Любую информацию компьютер воспринимает только в двоичной форме. Пользователь или программист взаимодействуют с компьютером через специальные графические интерфейсы: операционной системы, приложений. Программный код должен быть «переведен» на язык компьютера, т. е. *транслирован*. Если речь идет о компилируемых языках программирования, то исходный код с помощью программы компилятора преобразуется в специальный файл, который в дальнейшем становится программой (один или в совокупности с другими).

IDE обычно включает в себя:

- редактор кода;
- компилятор и/или интерпретатор;
- средства автоматизации сборки;
- отладчик.

Помимо перечисленного в IDE могут присутствовать и другие компоненты (например, конструкторы форм, анализаторы производительности, шаблоны проектов). Их совместная работа настраивается заранее автоматически либо предоставляется пользователю в удобном интерфейсе. Таким образом, зачастую проект достаточно скомпилировать и получить рабочее приложение.

Если мы говорим о языке ассемблера, то наиболее популярными IDE являются RadAsm, WinAsm.

Однако наряду с достоинствами IDE имеют весомые недостатки.

- Многофункциональные IDE требуют большой ресурс.
- Как правило, IDE добавляет в проект собственные файлы, необходимые только ей. Более того, форматы этих файлов могут обрабатываться лишь самой IDE и в случае переноса файлов программы на другой компьютер могут оказаться бесполезными.
- IDE могут быть «заточены» под конкретные операционные системы.
- Проекты могут компилироваться со стандартными опциями, которые не всегда требуются. Это отражается на размере исполняемого файла и даже скорости выполнения (впрочем в серьезных IDE опции можно менять).
- Многие IDE проприетарны.

Таким образом, IDE – лишь инструмент со своими достоинствами и недостатками. Существуют другие варианты разработок программ без использования указанных и иных IDE.

Ручная сборка и ее роль

Несложно понять, что до появления IDE программы собирались вручную, т. е. процесс компиляции осуществлялся в командной строке. С появлением IDE это стало делаться автоматически. Так для чего же необходимо знать основные приемы ручной сборки программы?

- Разработчик самостоятельно определяет опции компиляции. Это позволяет исключить создание ненужных файлов, уменьшить объем программ, увеличить скорость выполнения.

- Становится возможным более подробно изучить особенности языка программирования.
- Нет возможности использовать IDE, и ручная сборка – альтернативный вариант.

Конечно, большие проекты (например, Windows Forms) крайне неудобно разрабатывать вручную. Однако небольшие программы (а зачастую ассемблерные программы не требуют реализации сложных графических интерфейсов) удобно собирать именно вручную.

Автор пособия настоятельно рекомендует не использовать любые среды разработки в процессе обучения! Курс нацелен прежде всего на формирование фундаментальных знаний устройства программ и возможностей программиста. Описанные ниже алгоритмы и идеи распространяются на многие другие языки программирования.

Этапы компиляции ассемблерной программы

Для создания простой ассемблерной программы потребуется всего лишь два компонента:

- код программы;
- компилятор.

Код программы

Программный код можно набрать в любом текстовом редакторе. При этом нельзя использовать текстовые процессоры, такие как MS Word, поскольку форматирование текста может вносить дополнительные символы в код.

Самое простое средство – блокнот. Однако для постоянной работы лучше воспользоваться продвинутым редактором, включающим возможность подсветки синтаксиса используемого языка. Среди популярных редакторов с подсветкой синтаксиса и множеством других функций выделяют Notepad++, Sublime Text 3, Emacs, Vim (последние два ориентированы на опытных программистов). Файл ассемблерной программы имеет расширение **.asm**

Компилятор

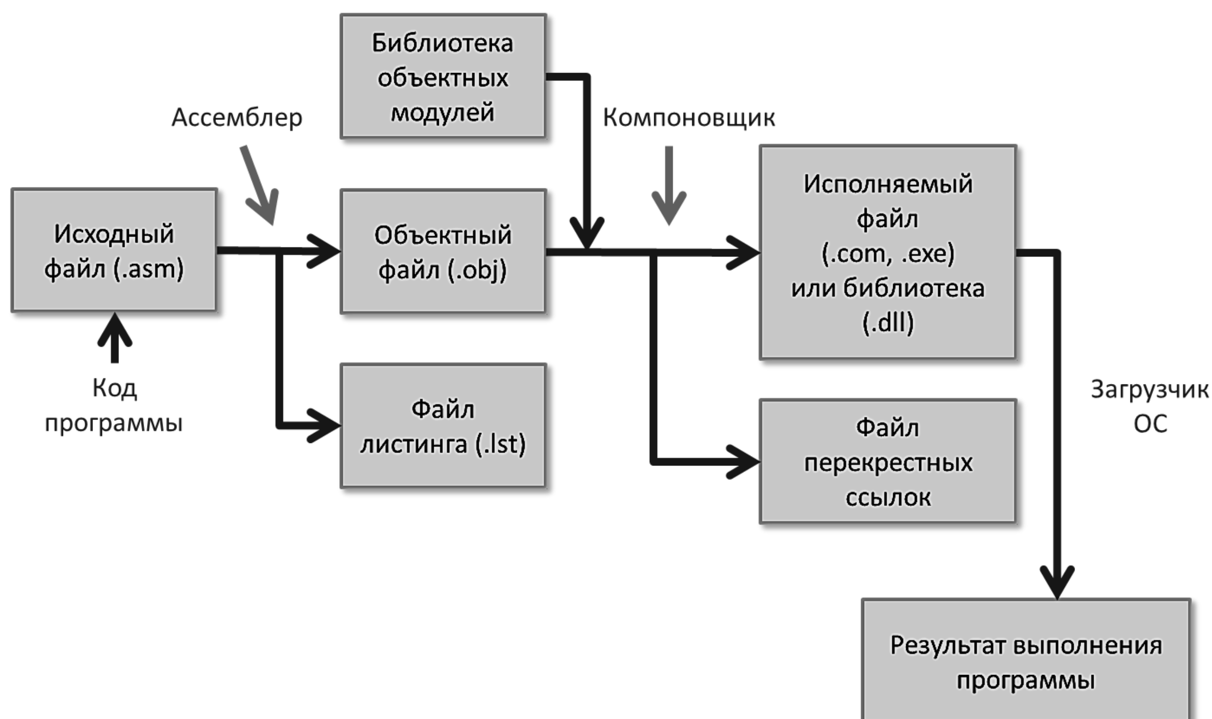
Разные ассемблеры – суть разные компиляторы.

Компиляция в Macro Assembler проходит в два этапа:

1) ассемблирование;
2) компоновка.

На этапе ассемблирования создается специальный объектный (бинарный) файл с расширением .obj. Этот файл транслирован в машинный язык.

Второй этап – компоновка, или линковка (linker – редактор связей). Задача линкера – собрать все созданные файлы в единую программу или библиотеку. Более подробно процесс компиляции можно представить в виде следующей схемы:

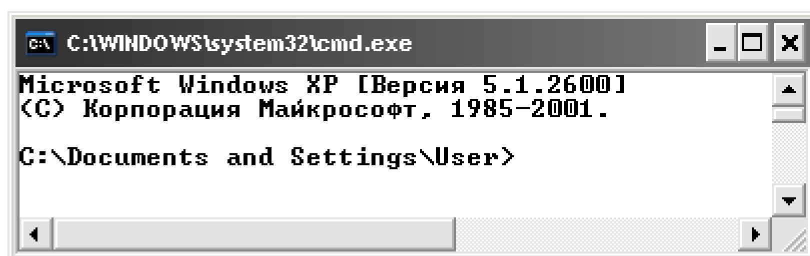


Командная строка

Командная строка *– программа с консольным интерфейсом, позволяющая пользователю с клавиатуры вводить команды управления работой компьютера.*

Вызвать командную строку можно несколькими способами:

- Пуск / Программы / Стандартные / Командная строка
- Пуск / Выполнить / cmd



Для компиляции достаточно уметь перемещаться по директориям и получать данные о файлах. Наиболее часто используются следующие команды:

HELP	отобразить список команд
HELP X	справка по конкретной команде X (help dir)
DIR	отображает содержимое директории
CHDIR путь	смена директории (chdir C:\Program Files)
CD ..	перейти в родительский каталог
CLS	очистка содержимого консоли
TREE	показать дерево файлов и каталогов, вложенных в данный
EXIT	выход
PAUSE	ожидать нажатия клавиши

У команд могут быть дополнительные атрибуты, причем необязательные обозначаются в квадратных скобках.

Пакетные файлы

Часто возникает необходимость выполнить несколько команд подряд. Вводить их вручную нерационально. Между тем, Windows предлагает рациональное решение этой проблемы – *пакетные файлы*.

Пакетный файл *– файл с командами, выполняемыми через командный интерпретатор операционной системы. В системе Windows они имеют расширение .bat*

Достаточно написать код команд, сохранить их в файл с расширением .bat и запустить его. При необходимости bat-файл редактируют. Например, чтобы удалить все файлы с форматом .txt и .docx в текущей папке, напишем в bat-файл

```
del *.txt *.docx
```

Использование пакетных файлов существенным образом упрощает процесс компиляции.

Алгоритм разработки программы

1. Для удобства создадим папку, в которую разместим проект.
2. Пишем код программы и меняем его расширение на .asm.

3. Ассемблируем код программы. Для этого в командной строке необходимо обратиться к транслятору и файлу, который компилируем. В случае удачного ассемблирования появится файл с расширением .obj.
4. Компонуем файл(ы). Для этого обращаемся к компоновщику и указываем ему полученный(ые) obj-файлы. Если компоновка прошла удачно, создается исполняемый файл (.com или .exe) или библиотека (.dll).

Формально в общем виде команда для ассемблирования следующая:

путь_к_компилятору\имя_компилятора [доп. опции] имя_программы.asm

а для компоновки:

путь_к_компоновщику\имя_компоновщика [доп. опции] имя_программы.obj

Компиляция MASM

Macro Assembler (MASM) – ассемблер, разработанный Microsoft для процессоров архитектуры Intel x86.

В текущей работе используется 11-я версия MASM, которая позволяет разрабатывать 16- и 32-битные приложения. Однако операционная система Windows 7 и выше не поддерживает работу с 16-битными приложениями как с устаревшим форматом; их можно запускать только с помощью DOS-эмуляторов (прил. 1), а разрабатывать соответственно с более ранних версий, например MASM 6.11[2].

Транслятором ассемблера является файл ml.exe, позволяющий транслировать 16- и 32-битные файлы. Если MASM 32 установлен на диске C, то полный путь к транслятору следующий:

C:\masm32\bin\ml.exe

Компоновщик зависит от разрядности создаваемой программы: для 16-битных это link16.exe, а для 32-битных – link.exe:

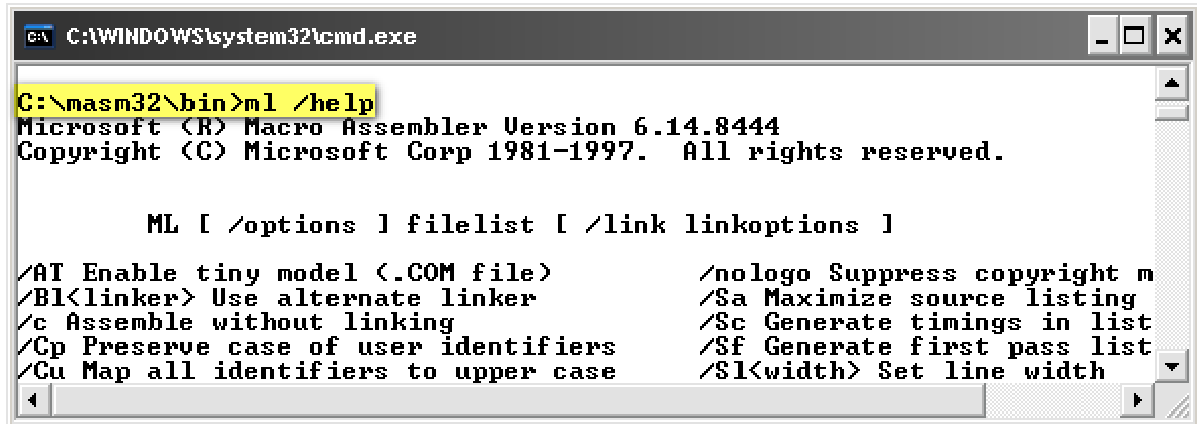
C:\masm32\bin\link16.exe

C:\masm32\bin\link.exe

---

[2] Разработка с помощью эмулятора обсуждается в прил. 1.

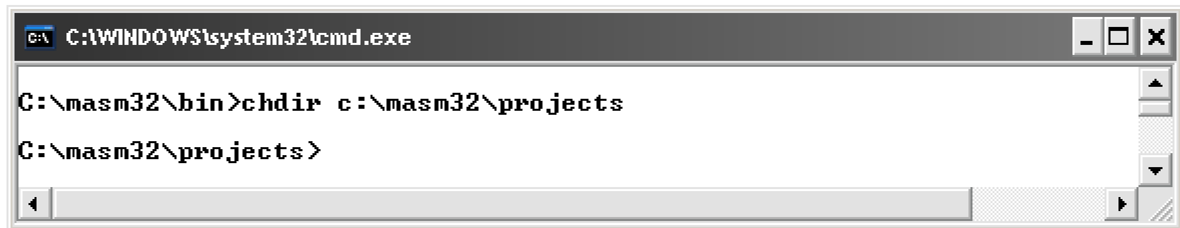
Чтобы просмотреть, какие опции есть у компилятора и компоновщика, воспользуйтесь опцией /help или /?



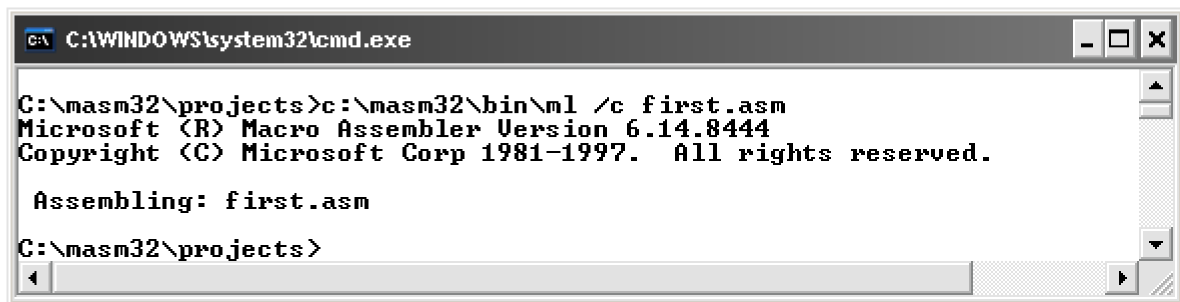
Скомпилируем 16-битное EXE приложение под MS-DOS. В искомой папке создаем файл с кодом, например:

```
.8086
.MODEL small
.STACK 100h

.DATA
Mess    BYTE    "Hello!$"

.CODE
start:
        mov    ax,@DATA
        mov    ds,ax

        mov    dx,OFFSET Mess
        mov    ah,09h
        int    21h

        mov    ah,08h
        int    21h

        mov    ah,4Ch
        int    21h
END    start
```

Вызываем командную строку и меняем директорию на папку с программой (чтобы не прописывать к ней полный путь)

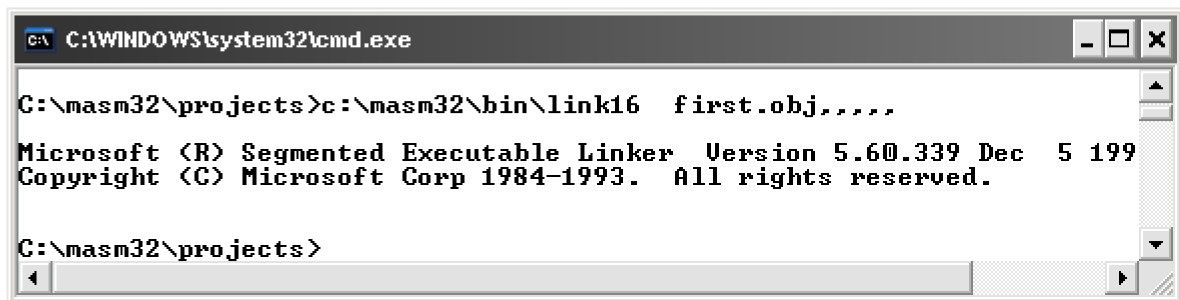


Для ассемблирования программы указываем путь к ml.exe, опцию /c и имя программы



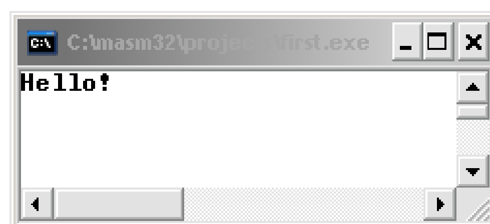
(обратите внимание, что необязательно прописывать ml.exe). Если возникли ошибки, то выведется предупреждение. Иначе в папке появится бинарный файл first.obj.

Для компоновки аналогично указываем путь к компоновщику (и опции при необходимости)



Подряд идущие запятые означают, что дополнительно создаваемые файлы будут иметь имя по умолчанию (в нашем случае first).

Если оба этапа прошли успешно, то в папке появится исполняемый файл программы first.exe



Очевидно, что для перекомпиляции потребуется повторить этот процесс. Поэтому в дальнейшем рекомендуется использовать bat-файл. Создадим bat-файл и поместим в него следующий код операций, которые мы проделали:

```
c:\masm32\bin\ml      /c first.asm
c:\masm32\bin\link16     first.obj,,,,,
pause
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Для каких целей создаются среды разработки?
2. Каковы основные преимущества использования среды разработки.
3. Чем полезна ручная сборка программы начинающему программисту?
4. Обозначьте основные инструменты и этапы разработки ассемблерной программы.
5. Какова роль командной строки при сборке программы? Для чего предназначены bat-файлы?
6. Почему в последних версиях Windows нет возможности напрямую создавать 16-битные приложения?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Запустите среду разработки RadAsm. Изучите ее основные компоненты.
2. Скомпилируйте рассмотренную в теории программу.
3. Запустите командную строку. Переместитесь в корневой каталог Windows и выведете его содержимое.
4. Создайте две папки. В первую поместите два текстовых файла. С помощью команды COPY скопируйте оба файла в другую папку.
5. Получите справку по опциями транслятора ml.exe и компоновщика link16.exe.
6. Скомпилируйте программу с помощью командной строки.
7. Напишите bat-файл, чтобы скомпилировать приведенную ранее программу автоматически.

Часть 2. MACRO ASSEMBLER: IA-16

Практическое занятие № 2.1

ОСНОВЫ MASM. ТИПЫ ДАННЫХ. КОМАНДЫ И ДИРЕКТИВЫ

1. ЦЕЛИ РАБОТЫ

- Изучить способы описания констант и идентификаторов.
- Установить различия между командами и директивами ассемблера.
- Изучить механизмы выделения памяти для переменных.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Константы

Целочисленная константа может содержать знак числа и суффикс системы счисления:

Суффикс	Система счисления
b	Двоичная
o или q	Восьмеричная
d (необязателен)	Десятичная
h	Шестнадцатеричная

Примеры констант:

101b, 01001110b
56o или 56q
48d или 48
3F12h, 0FFh

Если старший разряд шестнадцатеричной константы является буквой, то необходимо ставить перед ней нуль (как в последнем примере).

Вещественные константы используют в своей записи точку дробной части или показатель степени в экспоненциальной форме:

189.0 или 189.
32.08
-3.2E+04 (-32000)

Символьные константы окружаются одинарными или двойными кавычками:

'A' или "A"

Аналогично формируются и *строковые* константы:

'Macro Assembler.' или "Macro Assembler."

'Кавычки в "кавычках".' или "Кавычки в 'кавычках'."

Любой символ кодируется согласно принятому ASCII стандарту: на каждый символ отводится 1 байт памяти.

Идентификаторы

Идентификаторы для меток, переменных или констант не должны совпадать с зарезервированными словами. Первый символ может быть латинской буквой, символом подчеркивания, знаком @ или $, однако не может быть цифрой. Количество символов не должно превышать 247. Регистр букв по умолчанию не важен, однако может быть учтен при компиляции с опцией -Cp.

Команды и правило хорошего стиля

Команда *– оператор, выполняющийся в процессе работы программы (после компиляции).*

Любая команда в полной форме содержит четыре компоненты:

- метка (необязательна);
- команда;
- операнд(ы) (могут отсутствовать);
- комментарии (необязательны).

Строка может состоять также только из метки и/или комментария. Общая схема такова:

```
Метка:        Команда        Операнд(ы)        ;Комментарий
```

Имя метки завершается двоеточием. Однострочный комментарий начинается с точки с запятой.

Считается хорошим тоном соблюдать табуляцию между разделами. Пример блока кода:

```
Start:                              ; Метка входа
        mov   dx,OFFSET Mes         ; Загружаем адрес строки
        mov   ah,09h                ; Вызов функции 09h
        int   21h                   ; Прерывание ввода-вывода
```

Такое табулирование необязательно, однако подобный код хорошо читаем.

Директивы

***Директивы** – команды, влияющие на процесс компиляции.*

Директивы используются для определения сегментов памяти, описании данных, процедур и др. В отличие от команд ассемблера при дизассемблировании директивы не отображаются в качестве команд как таковых. Более того, различные ассемблеры поддерживают свои директивы, в чем-то схожие, либо, наоборот, специфичные для языка. В целом задача директив – упростить синтаксис программного кода.

Подробнее о метках, мнемонике команд, операндах и комментариях

***Метка** – это именованный участок ОЗУ. В процессе компиляции все имена меток заменяются на соответствующие адреса.*

Выделяют *метки кода* и *метки данных*.

Метки кода указывают участок программы, которому передается управление при условном или безусловном переходе:

```
Lbl_1:                  ; Метка
      ...
      jmp   Lbl_1       ; Безусловный переход на метку Lbl_1
```

Метка данных используется в качестве имени переменной, описываемой в сегменте данных:

```
Sum   WORD    0
Pi    REAL4   3.14
```

Мнемоника команд ассемблера построена на осмысленных именах, отображающих суть команды, например:

mov – move: двигать, перемещать

add – addition: сложение

jmp – jump: прыжок, переход

Команды могут содержать *операнды* (параметры). В качестве операндов допустимы регистры, переменные или константы. Константы могут выражаться в качестве простых и составных выражений, например:

```
mov   ax,3                ; в AX записать 3
mov   dx,(13 + 87)*45     ; в DX записать 4500
mov   bx,17 mod 4         ; в BX записать 1
```

Помимо однострочечных комментариев допустимы блочные, комментирующие произвольный фрагмент кода. Они описываются с помощью директивы COMMENT:

COMMENT * Комментирую некоторый блок кода *

Вместо символа * можно использовать любой другой.

Типы данных

В Macro Assembler определены 9 целочисленных и 3 вещественных типа, причем существует разделение на знаковые и беззнаковые.

Тип	Объем байт	Описание типа
BYTE	1	Беззнаковое целое
SBYTE	1	Знаковое целое
WORD	2	Беззнаковое целое
SWORD	2	Знаковое целое
DWORD	4	Беззнаковое целое
SDWORD	4	Знаковое целое
FWORD	6	Целое
QWORD	8	Целое
TBYTE	10	Целое
REAL4	4	Короткое вещественное
REAL8	8	Длинное вещественное
REAL10	10	Расширенное вещественное

Вещественные форматы соответствуют стандарту IEEE (стандарт представления вещественных чисел, Институт инженеров по электротехнике и электронике).

Диапазон возможных значений для каждого целого типа определяется по формулам:

- $0..2^k - 1$ для беззнаковых чисел;
- $-2^{k-1}..2^{k-1} - 1$ для знаковых чисел,

где k – разрядность типа (в битах).

Например, для типа BYTE ($k = 8$) диапазон допустимых значений 0..255, а для SBYTE $-128..127$.

В ранних версиях MASM использовались другие директивы описания, поддерживаемые до сих пор. Однако они не позволяют определить, к знаковому или беззнаковому числу относится переменная. Соответствие старых и новых директив приведено в таблице.

Новая версия	Старая версия
BYTE и SBYTE	DB
WORD и SWORD	DW
DWORD и SDWORD	DD
QWORD	DQ
TBYTE	DT
REAL4	DD
REAL8	DQ
REAL10	DT

Описание переменных

Синтаксис операции определения данных выглядит следующим образом:

[имя] тип инициализатор [, инициализатор] ...

Имя переменной «скрывает» собой адрес – смещение относительно начала сегмента, в котором она записана. Примеры описания переменных:

```
n    BYTE    235
m    BYTE    00100111b
k    SBYTE   -70
tmp  WORD    7AFEh
var1 QWORD   0FFFFFFFF00000000h
exp  REAL8   2.718281828
```

Независимо от указанной системы счисления для константы ассемблер обрабатывает ее в двоичной форме.

Ассемблер допускает *множественную инициализацию*: одной метке назначается несколько ячеек памяти указанного типа, которые инициализируются значениями:

```
.data                         ; сегмент данных
M1   BYTE    48h, 0FFh, 10h
M2   WORD    256, 1001b, 77o, 3A6h
M3   SBYTE   -1, -2, -3, -4, -5
     SBYTE   -6, -7, -8, -9, -10
     SBYTE   -11, -12
```

Для переменной M1 отводится три байта памяти, для M2 – 8 байт, для M3 – 12 байт. Очевидно, что общий объем отводимой памяти вычисляется по формуле

$$\text{объем памяти} = \text{количество значений} \times \text{размер типа}.$$

Предположим, что начальное смещение в сегменте данных равно нулю. Тогда согласно правилам адресации памяти процессоров x86 компилятор распределит память для переменных по следующей схеме:

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	...	0016
48	FF	10	00	01	09	00	3F	00	A6	03	...	F4
48h	0FFh	10h	256		1001b		77o		3A6h		...	-12h
M1			M2								M3	

В первой строке обозначены адреса; во второй – значения байт по указанным адресам; в третьей – исходные числа. Особое внимание обратите на порядок записи и конвертацию (например, значение –12h представляется как F4h согласно дополнительному коду для отрицательных чисел).

Оператор повторения DUP

Оператор DUP создает последовательность дублируемых блоков памяти заданного типа. Он часто используется при выделении памяти под строку символов или массив:

```
Mass  BYTE 10  DUP(0)           ; требуется 10Б памяти
Arr   WORD 20  DUP(23h,0h,0AEh)  ; требуется 120Б памяти
Mes   BYTE 100 DUP("Hello ")     ; требуется 600Б памяти
```

Так, переменной Mass выделяется 10 байт памяти, каждый байт изначально обнуляется. Переменная Arr – последовательность 20 копий слов 23h, 0h, 0AEh. Переменная Mes ссылается на 100 подряд идущих копий указанного текста (пробел тоже кодируется байтом!).

Инициализированные и неинициализированные переменные

При описании переменную необязательно инициализировать значением: можно допустить, что значение будет присвоено переменной в процессе работы программы. Тогда в качестве инициализатора выступает символ «?». Неинициализированными могут быть и последовательности байтов. Также допускается комбинирование инициализированных и неинициализированных переменных.

Например:

```
M1   WORD   ?
M2   BYTE    ?,?
M3   QWORD  ?,1000h,2000h,?,?
```

Неинициализированные переменные рекомендуется описывать в секции .data? (в отличие от инициализированных, описываемых директивой .data). Это связано с тем, что во время компиляции переменным выделяются требуемые объемы памяти, информация о которых включается в исполняемый файл. Так, объявив переменную

```
Mass   TBYTE   100000  DUP(?)
```

исполняемый файл увеличивается почти на 1МБ! Это слишком расточительно выделять такой объем памяти под пустые ячейки. Гораздо рациональнее использовать оперативную память при непосредственном выполнении программы, что и делает директива .data? Ниже приведен пример рационального описания переменных:

```
.data
Arr      DWORD   100     DUP(0)
n        SBYTE   -23

.data?
Mass     TBYTE   100000  DUP(?)
m        SBYTE   ?
```

Директива присваивания

Директива присваивания «=» связывает метку с целочисленным выражением. Синтаксис директивы:

```
имя = выражение
```

Выражение является целым числом, укладывающимся в промежутки 32-разрядного. Подобные метки необходимы для создания более гибкого кода. Так, в следующем примере установленное значение для метки MyVal подставляется во все вызывающие ее операторы:

```
MyVal = 2015
mov   ax,MyVal
add   bx,MyVal
push  MyVal
```

На стадии *препроцессирования*[3] метка MyVal заменяется непосредственным значением 2015. В процессе метке можно неоднократно присвоить другое значение.

Счетчик команд $

При описании массивов или строк почти всегда необходимо знать их размер (количество элементов). Ручное сопровождение – не лучшее решение проблемы.

MASM предлагает специальный оператор $, возвращающий текущее значение счетчика команд, т. е. смещение текущего оператора относительно начала сегмента.

Пусть определены массив и оператор $:

```
Arr      BYTE    1,2,3,4,5
LenArr   =       $-Arr
```

Если, например, переменная Arr имеет адрес (смещение), равное 10h, то оператор-счетчик $ хранит значение 0015h, откуда разность $-Arr дает 5 – количество элементов массива:

...	0010h	0011h	0012h	0013h	0014h	0015h	...
...	01	02	03	04	05	$	...

В случае, когда размер элементов равен нескольким байтам, разность необходимо разделить на размер типа:

```
Arr      DWORD   1,2,3,4,5
LenArr   =       ($-Arr)/4
```

Оператор $ должен идти сразу после искомого массива. В противном случае результат окажется неверным.

Впрочем, для подобного рода задач есть несколько замечательных директив (о них поговорим при исследовании циклов).

Директива EQU

Директива используется для назначения символьного имени целочисленному выражению или произвольной текстовой строке. Существуют три формы директивы EQU:

---

[3] Препроцессирование – замена меток, директив, макросов и прочего на соответствующие значения и блоки кода.

```
имя   EQU   выражение
имя   EQU   символ
имя   EQU   <текст>
```

В качестве выражения можно брать целое число. Символ – определенное ранее выражение с помощью директивы = или EQU. В третьем случае определяется текстовое выражение или вещественное число.

Таким образом, директива EQU имеет схожий функционал с директивой присваивания =, распространяющийся не только на целочисленные выражения. Однако EQU не допускает переопределения метки новым значением.

Например:

```
M   =   20
N         EQU   10       ; N:=10
W         EQU   M        ; W:=M (т. е. W:=20)
KeyPress  EQU   <"Нажмите любую клавишу...",0>
                         ; KeyPress:="Нажми...", 0
Pi        EQU   <3.14>   ; Pi:=3.14

.data
; Key:= "Нажмите любую клавишу...",0
Key   BYTE  KeyPress
Mass  WORD  N  DUP(0)  ; равносильно Mass WORD 10 DUP(0)
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Сколько систем счисления доступно в MASM?
2. Перечислите директивы для описания типов переменных. Чем отличаются знаковые типы от беззнаковых?
3. Почему директивы не отображаются при дизассемблировании программы?
4. Как можно использовать оператор dup при создании массива?
5. В чем различия между инициализироавнными и неинициализированными переменными? Для чего используется директива «.data?»?
6. Что представляет собой счетчик команд и чем он полезен в программе?
7. Опишите различия между директивой = и оператором EQU.

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Описать переменную размером в слово и переменную размером в учетверенное слово со знаком. Опишите и инициализируйте значениями массив типа байт из четырех чисел, где каждое число представлено в двоичной, восьмеричной, десятичной и шестнадцатеричной системах счисления соответственно.
2. Определить, сколько байт памяти занимают следующие переменные:
   - M DWORD 45
   - N WORD 23, FFh
   - Arr BYTE "Hello, world!"
   - Mass BYTE 100 DUP(23, " Assembler ", 0, 45h)
3. Какие ошибки допущены при описании следующих переменных:
   - Text WORD "Hello, world"
   - N BYTE 260
   - M DWORD 24, FFh, 45h
   - K BYTE -5
   - Arr WORD DUP(76, 34)
4. Объяснить, почему в следующем блоке кода метка kol неверно возвращает число элементов массива:

```
Mass   DWORD   34, 567, -230, 0, 123, 2015
kol    =       ($-Mass)/2
```

Практическое занятие № 2.2

ШАБЛОН КОНСОЛЬНОГО ПРИЛОЖЕНИЯ. ЗНАКОМСТВО С ОТЛАДЧИКОМ

1. ЦЕЛИ РАБОТЫ

- Рассмотреть основные компоненты 16-битного приложения.
- Познакомиться с основными элементами отладчика Turbo Debugger.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Шаблон 16-битного консольного приложения

В этой части практикума рассматриваются только 16-битные приложения, ориентированные на систему MS-DOS. Несмотря на то что указанная операционная система давно устарела, а ее программы на современных операционных системах зачастую запускаются только с DOS-эмуляторов, изучение таких программ весьма важно для понимания механизмов работы ассемблерного кода.

Шаблон 16-битного консольного приложения:

```
.8086                       ; Режим работы процессора
.model small                ; Модель памяти
.stack 100h                 ; Размер стека

.const                      ; Раздел описания констант

.data                       ; Сегмент данных

.data?                      ; Сегмент неинициали-
                            ; зированных данных
.code                       ; Сегмент кода
Start:                      ; Метка входа
        mov   ax,@data      ; В регистр DS сохраняем
        mov   ds,ax         ; адрес сегмента данных

        ;;  Код программы  ;;

        mov   ah,4Ch        ; Завершение процесса
        int   21h
end     Start
```

Для компиляции приложения напишите bat-файл:

```
c:\masm32\bin\ml   /c   shablon.asm
c:\masm32\bin\link16    shablon.obj,,,,,
pause
```

Некоторые команды подробнее будут рассмотрены на следующих занятиях.

Директива .8086 задает режим работы процессора, она ограничивает нас набором инструкций процессора Intel 8086. В данном типе приложений нам не требуется большего.

Директива .model устанавливает модель памяти. Модель small характеризует реальный режим адресации, причем для данных (переменных), стека и программного кода выделяются разные сегменты памяти. Это основной режим 16-битного приложения, дающий исполняемый файл .exe.

Директива .stack резервирует указанное число байт для стековых операций. В данном случае это 100h = 256 байт.

Следующие три директивы задают разделы описания констант и переменных.

Директива .code определяет сегмент, в котором записана информация о нашей программе.

Конструкция Start – end Start определяет область реализации основного программного кода. Метку Start (имя можно задать любым, но незарезервированным) часто называют *точкой входа* в программу.

Команды

```
mov    ax,@data
mov    ds,ax
```

записывают в регистр DS адрес сегмента данных. Для модели small эти команды обязательны, если используются переменные. В противном случае данные могут читаться совершенно из другого участка памяти!

Созданное приложение лишь запускает консоль и сразу же ее закрывает. Все остальное, от задержки консоли до ввода и вывода символов, строк, чисел, требуется реализовывать практически с «нуля»!

Из шаблона можно убрать разделы, которые в нем не используются, например:

```
.8086                    ; Режим работы процессора
.model small             ; Модель памяти

.code                    ; Сегмент кода
Start:                   ; Метка входа
      ;; Код программы ;;
      mov   ah,4Ch        ; Завершение процесса
      int   21h
end    Start
```

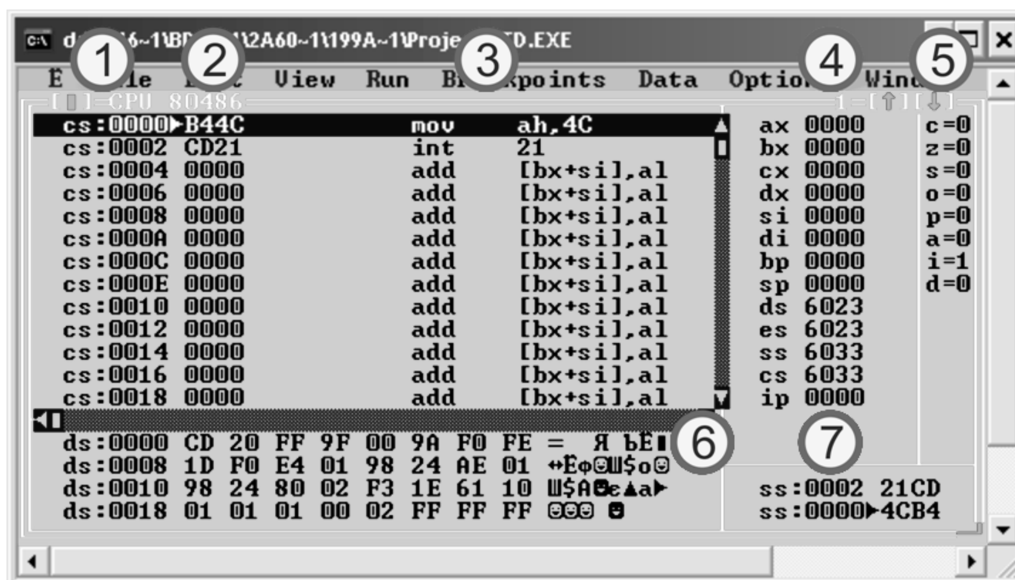
Отладка программы

Для эффективного изучения ассемблера и поиска ошибок важно развить умение работы в отладчике.

Поскольку реальный и защищенный режимы работы процессора имеют существенные отличия в схеме адресации, то и отладчики ориентированы на каждый из них. Рассмотрим 16-битный отладчик от компании Borland Turbo Debugger.

Turbo Debugger (TD) – один из наиболее мощных для своего времени отладчиков, предназначенный для дизассемблирования и отладки программного кода. TD предназначался для .exe и .com приложений в операционной системе MS-DOS. Впрочем, в TASM 5 был добавлен и отладчик для 32-битной архитектуры.

После запуска TD.exe появится окно отладчика. Откроем в нем .exe файл вышеуказанной программы:



1. Адресное пространство программного кода. За сегмент кода отвечает регистр CS.
2. Опкоды инструкций в оперативной памяти.
3. Инструкции процессору.
4. Значение регистров в текущий момент времени.
5. Значение бит у регистра флагов.
6. Дамп памяти. Левая колонка задает адрес. Центральная выводит содержимое байт памяти (в строке по 8 байт). Правая содержит символы опкода с точки зрения ASCII представления.
7. Содержимое стека.

Из первого и второго столбцов можно узнать, сколько байт памяти занимает определенная команда: достаточно найти разность между адресами или посчитать количество байт в опкоде. Черная полоса является курсором.

Небольшой треугольник между первым и вторым столбцом – указатель команды, готовой к выполнению. Известно, что адрес выполняемой команды хранится в регистре IP, а весь адрес задается парой CS:IP. В нашем примере IP = 0 (что подтверждается колонкой 4).

Обратите внимание, что код отображается не в точности, как в нашей программе:

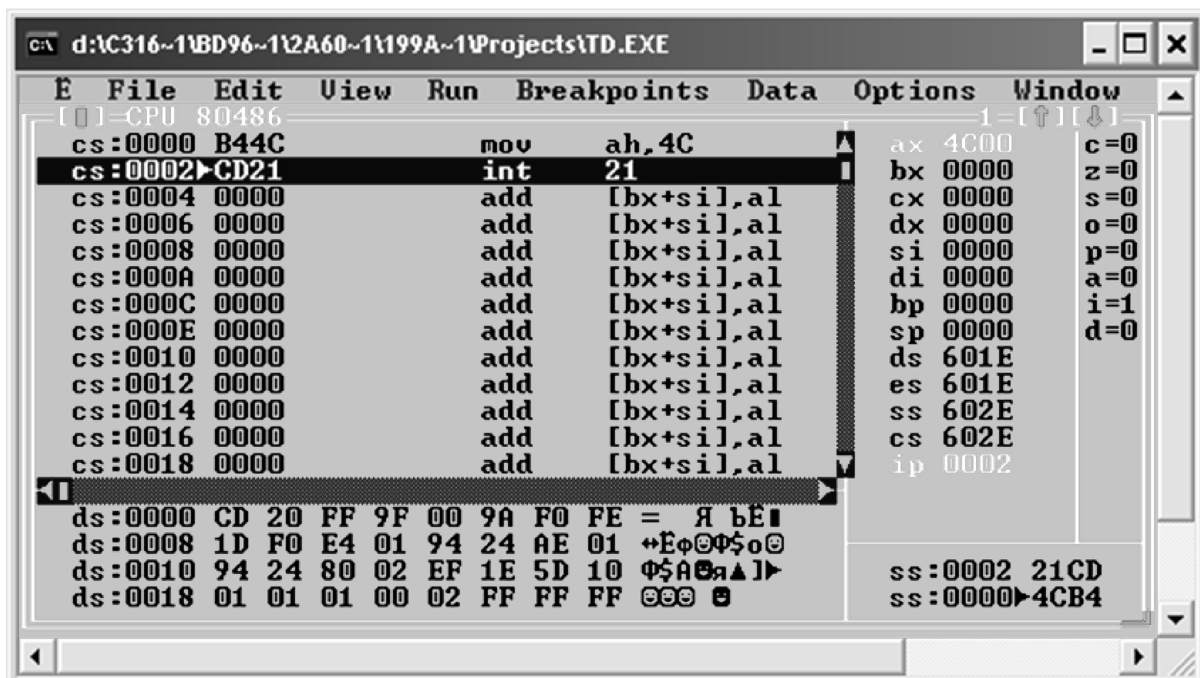
```
mov   ah,4Ch
int   21h
```

Это говорит о том, что другие команды являются директивами.

Пошаговое выполнение (трассировку) можно выполнять на различных уровнях:

- [F7] – выполняется одна команда;
- [F8] – выполняется одна команда, причем процедуры и циклы считаются одной командой;
- [F4] – выполнять до ближайшей контрольной точки (для выставления контрольной переведите курсор на команду и нажмите F2);
- [F9] – выполнить всю программу до конца.

Выполним первую команду, нажав клавишу F7.



По столбцу регистров можно заметить, что изменилось значение AX и IP. Указатель команды сместился на следующую.

Преобразуем программу, добавив несколько переменных:

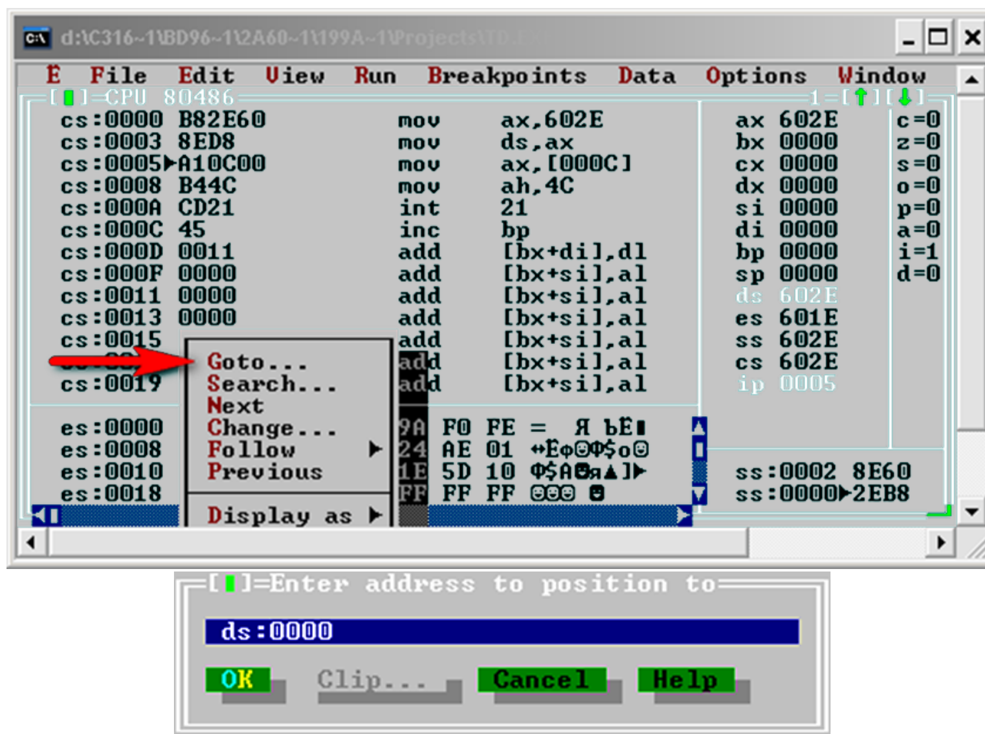
```
.8086
.model small

.data
A   word   45h
B   byte   11h

.code
Start:
        mov    ax,@data
        mov    ds,ax

        mov    ax,A

        mov    ah,4Ch
        int    21h
end    Start
```

В результате получим следующее:

```
cs:0000►B82E60          mov     ax,602E
cs:0003 8ED8            mov     ds,ax
cs:0005 A10C00          mov     ax,[000C]
cs:0008 B44C            mov     ah,4C
cs:000A CD21            int     21
```

Директива @data заменяется на номер сегмента, в котором будут записаны данные (переменные). Вместо переменной A указан ее адрес – смещение внутри сегмента данных (000C).

Посмотрим, как переменные A и B расположились в сегменте данных. Если дамп памяти не адресуется парой DS:XXXX, то выделим его и осуществим переход в начало сегмента



Обратим внимание на дамп. Наши переменные записаны следующим образом:

```
ds:0000 B8 2E 60 8E D8 A1 0C 00
ds:0008 B4 4C CD 21 45 00 11 00
ds:0010 00 00 00 00 00 00 00 00
ds:0018 00 00 00 00 00 00 00 00
```

Тогда переменная A имеет адрес DS:000C, переменная B – DS:000E.

В следующих пунктах мы продолжим изучение отладчика.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. С какими проблемами могут столкнуться разработчики 16-битных приложений, использующие последние версии Windows? Как их решить?
2. Перечислите ключевые директивы и команды консольного приложения?
3. Какие элементы программы позволяет отслеживать отладчик Turbo Debugger?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

Опишите переменную типа word, две переменные типа dword и переменную типа byte. С помощью отладчика найти эти переменные и указать их адреса.

Практическое занятие № 2.3

КОМАНДЫ ПЕРЕСЫЛКИ ДАННЫХ И АРИФМЕТИЧЕСКИЕ КОМАНДЫ

1. ЦЕЛЬ РАБОТЫ

- Изучить команды пересылки данных и арифметические команды.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Команда пересылки данных mov

```
mov    приемник, источник
```

Команда mov копирует данные из источника в приемник. Источник может быть регистром, переменной или значением, а приемник – регистром или переменной. Равносильна команде присваивания и имеет следующие ограничения:

- размеры операндов должны совпадать;
- один из операндов обязан быть регистром;
- операндами-приемниками не могут быть регистры IP, CS;
- непосредственное значение запрещено пересылать сегментному регистру напрямую.

Например:

```
mov    ax,340     ; AX := 340
mov    bl,4       ; BL := 4
mov    tmp,ax     ; tmp:= AX; tmp имеет тип WORD / SWORD
mov    bl,ax      ; ошибка! Размеры операндов не совпадают!
```

Многие команды ассемблера с несколькими операндами, как правило, требуют, чтобы последние имели одинаковый размер.

Задача 1

Написать приложение, копирующее значение из одной переменной в другую.

```
.8086
.model small
.stack 100h
```

```
.data
X       WORD   2015
Y       WORD   0

.code
Start:
        mov    ax,@data
        mov    ds,ax

        mov    ax,X       ; AX:=X
        mov    Y,ax       ; Y:=AX  (т. е. Y:=X)

        mov    ah,4Ch
        int    21h
end    Start
```

Команда обмена данных xchg

```
xchg    операнд_1,операнд_2
```

Команда меняет местами содержимое операндов. Операндами могут быть регистры или переменные. Размеры операндов должны совпадать.

Примеры:

```
xchg   ax,bx
xchg   cx,tmp     ; переменная tmp имеет тип WORD / SWORD
xchg   ah,al      ; обмен старшего и младшего байта в AX
xchg   bx,al      ; ошибка! Разный размер операндов
```

Задача 2

Написать приложение, обменивающее две переменные значениями.

Команда xchg не может работать с двумя переменными одновременно, поэтому используем для хранения промежуточного результата регистр AX:

```
.8086
.model small
.stack 100h
```

```
.data
X      WORD   2015
Y      WORD   0

.code
Start:
       mov    ax,@data
       mov    ds,ax

       mov    ax,X         ; AX:=X
       xchg   ax,Y         ; AX:=Y, Y:=AX (т. е. Y:=X)
       mov    X,ax         ; X:=AX (т. е. X:=Y)

       mov    ah,4Ch
       int    21h
end    Start
```

Арифметические команды

К базовым арифметическим операциям ассемблера относятся команды сложения, разности, умножения и деления. Каждая команда может иметь вариацию, предполагающую дополнительную функцию. Команды сложения и разности имеют похожий синтаксис, а умножения и деления – весьма специфические особенности.

Сложение	**Разность**
add / adc / inc	sub / sbb / dec
Умножение	**Деление**
mul / imul	div / idiv

Команды add, adc, inc

add   приемник, источник

Команда add складывает источник с приемником и записывает результат в приемник. Приемник может быть регистром или переменной, источник – регистром, переменной или константой. Операнды должны иметь одинаковый размер.

Примеры:

```
add   ax,bx     ; AX := AX + BX
add   bx,tmp    ; BX := BX + tmp
add   ch,30     ; CH := CH + 30
add   tmp,al    ; tmp:= tmp + AL
```

adc   приемник, источник

Команда adc складывает источник с приемником и с флагом переноса CF, результат записывается в приемник. Аналогична команде add, однако способна учитывать перенос.

inc   операнд

Увеличивает операнд на единицу (флаг переноса CF не меняет!). Операнд может быть регистром или переменной. Работает быстрее add/adc и кодируется всего одним байтом.

Команды sub, sbb, dec

sub   приемник, источник

Команда sub вычитает источник из приемника и результат записывает в приемник. Приемник может быть регистром или переменной, источник – регистром, переменной или константой. Операнды должны иметь одинаковый размер.

Примеры:

```
sub   ax,bx     ; AX := AX – BX
sub   bx,tmp    ; BX := BX – tmp
sub   ch,30     ; CH := CH – 30
sub   tmp,al    ; tmp:= tmp – AL
```

sbb   приемник, источник

Команда sbb вычитает из приемника источник и флаг переноса CF, результат записывается в приемник. Аналогична команде sub, однако способна учитывать перенос.

dec   операнд

Уменьшает операнд на единицу (флаг переноса CF не меняет!). Операнд может быть регистром или переменной. Работает быстрее sub/sbb и кодируется всего одним байтом.

Задача 3

Написать приложение, вычисляющее $2x - y + 1$ *при заданных* x *и* y *размером в слово (предполагается, что переносов не возникает).*

```
.8086
.model small
.stack 100h

.data
X      SWORD  10h
Y      SWORD  20h
Rez    SWORD  ?        ; переменная для записи результата

.code
Start:
       mov    ax,@data
       mov    ds,ax

       mov    ax,X
       add    ax,ax   ; AX := AX + AX (равносильно x+x=2x)
       sub    ax,Y    ; AX := AX - Y (AX := 2x - y)
       inc    ax      ; AX := AX + 1

       mov    Rez,ax   ; записываем результат в
                       ; новую переменную
       mov    ah,4Ch
       int    21h
end    Start
```

Команды mul и imul

mul операнд

Операнд – это только один из сомножителей. Он может находиться в регистре или в памяти, второй сомножитель – это содержимое регистра AL либо AX. Под результат отводится в два раза больше места, чем под сомножители и его местонахождение зависит от типа сомножителей.

- При умножении байтов результат имеет размер слова и записывается в регистр AX.

- При умножении слов результат имеет размер двойного слова и записывается в пару регистров DX:AX (AX – младшее слово, DX – старшее).

Примеры:

```
; Умножение байта на байт
mov   al,10h
mov   bl,30h
mul   bl          ; AX := AL * BL  (AX = 0300h)

; Умножение слова на слово
mov   ax,2000h
mov   cx,500h
mul   cx          ; DX:AX := AX * CX (DX:AX = 00A0:0000h)
```

Чтобы «собрать» число из пары DX:AX и записать его в переменную размером в 4 байта, необходимо использовать косвенную адресацию.

```
imul    операнд
```

Команда аналогична команде mul. Работает для чисел со знаком.

В отличие от mul на результат отводится в два раза больше памяти только при необходимости.

Задача 4

Написать приложение, вычисляющее $y = (kx + 1)^2$ *при заданных* x *и* k *размером в байт. Предполагается, что* $kx + 1$ *умещается в байт, а его квадрат может и превосходить.*

```
.8086
.model small
.stack 100h

.data
X       BYTE   10h
k       BYTE    3h
Y       WORD    ?
```

```
.code
Start:
        mov   ax,@data
        mov   ds,ax

        mov   al,X   ; AL := x
        mul   k      ; AX := AL * k (kx)
        inc   al     ; AL := AL + 1 (kx + 1)
        mul   al     ; AX := AL * AL ([kx + 1]^2)

        mov   Y,ax

        mov   ah,4Ch
        int   21h
end   Start
```

Обратите внимание, что после первого умножения результат записывается целиком в AX! Однако мы предполагаем, что результат не больше байта, поэтому берем лишь младшую часть – AL (тогда как старшая AH = 0), не нарушая тем самым корректность.

Во втором случае нас уже интересует весь AX, так как в AL может не поместиться все значение. Иначе возможна потеря данных.

Команды div и idiv

div   операнд

Деление слова на байт или двойного слова на слово. Местонахождение делимого строго фиксировано – регистр AX либо пара DX:AX. Операнд является делителем, в два раза меньше делимого (по объему памяти).

Результатом будут два числа – неполное частное и остаток. Остаток сохраняется в старшую часть, неполное частное – в младшую.

- Если делимое в AX, то в AH запишется остаток, в AL – неполное частное.
- Если делимое в DX:AX, то в DX запишется остаток, в AX – неполное частное.

Важно следить, чтобы частное или остаток умещалось в результат. В противном случае возникает ошибка «Деление на ноль».

Примеры:

```
; Делим 1000h на 20h
mov   ax,1000h
mov   bl,20h
div   bl              ; AH = 00h, AL = 80h

; Делим 12022h на 60h
mov   ax,2022h
mov   dx,0001h   ; DX:AX = 0001:2022h
mov   bx,0060h
div   bx              ; DX = 0022h, AX = 0300h
```

idiv   операнд

Команда аналогична команде div. Работает для чисел со знаком.

Задача 5

Написать приложение, вычисляющее неполное частное и остаток выражения uv/w *при заданных* u, v *и* w *размером в байт.*

```
.8086
.model small
.stack 100h

.data
u      BYTE   10
v      BYTE   15
w      BYTE   3

.data?
ost    BYTE   ?
nep    BYTE   ?

.code
Start:
       mov    ax,@data
       mov    ds,ax

       mov    al,u       ; AL := u
       mul    v          ; AX := AL * v (u * v)
       div    w          ; AL := неполное частное
                         ; AH := остаток
```

```
        mov    ost,ah
        mov    nep,al

        mov    ah,4Ch
        int    21h
end     Start
```

Команда neg

neg операнд

Изменяет знак операнда на противоположный. Операнд может быть регистром или переменной.

Примеры:

```
mov  ax,34
neg  ax          ; AX := -34
mov  bl,-80h     ; BL : = -80h (-128)
neg  bl          ; Но знак не изменится, поскольку 128
                 ; не умещается в байт со знаком
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Опишите работу операторов mov и xchg.
2. Каким образом осуществляются арифметические операции в ассемблере?
3. В каком случае при использовании команды neg не происходит изменения знака операнда размером в слово?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Изучите основы работы с отладчиком Turbo Debugger, используя прил. 2.
2. Опишите две переменные размером в слово каждая. Первой присвоить значение регистра AX, второй – сумму значений регистров BX и DX.
3. Опишите три переменные размером в слово. Сложите первые две и вычтите третью. Результат запишите в первую переменную.
4. Вычислите $x^2 - 6x + 9$ при заданном x за наименьшее число изученных операций.
5. Разделите число $23000\mathrm{F}_{16}$ на 2000_{16}. Неполное частное и остаток запишите в отдельные переменные.

Практическое занятие № 2.4

КОМАНДЫ УСЛОВНОГО И БЕЗУСЛОВНОГО ПЕРЕХОДА

1. ЦЕЛЬ РАБОТЫ

- Изучить команды условных и безусловных переходов.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Организация условных переходов в ассемблере отличается от конструкций, применяемых в языках программирования высокого уровня.

В процессе условного перехода можно выделить два этапа:

- в ходе некоторых операций устанавливаются биты в регистре флагов;
- согласно значениям определенных флагов осуществляется (либо нет) переход.

Команда jmp

```
jmp    метка
```

Команда предназначена для безусловного перехода на участок кода, указанный меткой. Метка должна завершаться двоеточием.

Выделяют *локальные* и *глобальные* метки. Локальные имеют область видимости только внутри процедур, а глобальные используются для организации связи между процедурами. Обычно глобальных переходов стоит избегать, поскольку можно нарушить логическую целостность программы.

Пример:

```
    . . .
    jmp   Exit      ; Перейти на метку Exit
    . . .
 Exit:
    . . .
```

Важно понимать, что метка – именованный адрес, т. е. «Exit» скрывает под собой число – адрес участка кода, на который совершается переход. В этом можно убедиться, воспользовавшись отладчиком.

Команда cmp

cmp операнд_1,операнд_2

Команда меняет значение регистра флагов в зависимости от величины разности между первым и вторым операндом. Как правило, она предваряет условный переход.

Фактически эта команда сравнивает два операнда. Например, при сравнении беззнаковых чисел по значениям флага нуля ZF и переноса CF можно судить, в каком отношении находятся операнды.

ZF	CF	Отношение
0	1	операнд_1 < операнд_2
0	0	операнд_1 > операнд_2
1	0	операнд_1 = операнд_2

Для чисел со знаком дополнительно потребуется контроль флага знака SF.

Например, проанализируем следующий код:

```
mov   ax,34h
mov   bx,44h
cmp   ax,bx
```

```
mov   ax,602C      ax 0034   c=1
mov   ds,ax        bx 0044   z=0
mov   ax,0034      cx 223C   s=1
mov   bx,0044      dx 0034   o=0
cmp   ax,bx        si 0330   p=1
```

В процессе выполнения команды сравнения был установлен флаг CF, а ZF не изменился. Если считать сравниваемые величины беззнаковыми, то согласно таблице они находятся в отношении «<» (что является истиной).

Команды условного перехода j*

По существу эти команды являются расширением команды jmp. Однако до перехода проверяется состояние флагов, закрепленных за командой перехода. Если флаги принимают требуемые значения, то переход осуществляется по указанной метке; в противном случае выполняется следующая команда.

Команды условного перехода можно разделить на три группы:

- переход после сравнения знаковых либо беззнаковых чисел;
- переход по состоянию определенных флагов;
- переход по состоянию регистра CX.

Команды перехода для знаковых и беззнаковых чисел

Ставятся после команды сравнения

cmp a, b

Команда	Условие перехода на метку	Состояние флагов для перехода
Для любых чисел		
JE метка	a = b	ZF = 1
JNE метка	a <> b	ZF = 0
Для чисел со знаком		
JL/JNGE метка	a < b	SF <> OF
JLE/JNG метка	a < = b	SF <> OF или ZF = 1
JG/JNLE метка	a > b	SF = OF и ZF = 0
JGE/JNL метка	a > = b	SF = OF
Для чисел без знака		
JB/JNAE метка	a < b	CF = 1
JBE/JNA метка	a < = b	CF = 1 или ZF = 1
JA/JNBE метка	a > b	CF = 0 и ZF = 0
JAE/JNB метка	a > = b	CF = 0

Команды перехода по состоянию флагов

Команда	Состояние флагов для перехода
JZ метка	ZF = 1
JS метка	SF = 1
JC метка	CF = 1
JO метка	OF = 1
JP метка	PF = 1
JNZ метка	ZF = 0
JNS метка	SF = 0
JNC метка	CF = 0
JNO метка	OF = 0
JNP метка	PF = 0

Команда перехода по состоянию регистра CX

Команда jcxz переходит на указанную метку, если CX = 0.

Задача 1

Даны два числа размером в слово. Написать программу, записывающую в регистр AX максимальное из этих чисел при условии:

- *оба числа беззнаковые;*
- *оба числа со знаком.*

Для беззнаковых чисел:

```
.8086
.model small
.stack 100h

.data
a     word   26h
b     word   25h

.code
Start:
      mov    ax,@data
      mov    ds,ax

      mov    ax,a      ; Допустим, что максимум - a
      cmp    ax,b      ; Сравниваем числа и устанавливаем
                       ; флаги
      jae    lbl_End   ; a >= b? Если да, то переход на метку
      mov    ax,b      ; иначе b больше
lbl_End:
      mov    ah,4Ch
      int    21h

end   Start
```

Для чисел со знаком достаточно изменить тип переменных

```
a     sword   -26h
b     sword   25h
```

и операцию перехода

```
      jge    lbl_End
```

Задача 2

Даны два числа a и b. Требуется определить, принадлежит ли заданное число x отрезку $[a, b]$*?*

Поскольку вывод текста в ассемблере еще не рассматривался, опишем переменную флажок, равную нулю, если точка не принадлежит отрезку и единице, если принадлежит:

```
.8086
.model small
.stack 100h

.data
a     word   10h
b     word   20h
x     word   1Ah
flag  byte   0h        ; 0 – не принадлежит, 1 - принадлежит

.code
Start:
      mov    ax,@data
      mov    ds,ax

      mov    ax,x
      cmp    ax,a
      jb     @exit     ; x < a? Тогда выход
      cmp    ax,b
      ja     @exit     ; x > b? Тогда выход
      inc    flag      ; иначе x принадлежит [a,b]
@exit:

      mov    ah,4Ch
      int    21h

end   Start
```

Обратите внимание, что условия проверяются от противоположного. В противном случае придется добавить новые метки и безусловные переходы, что увеличит код.

В ассемблере зачастую условия рациональнее проверять от противоположного. Это уменьшает объем необходимых инструкций.

Задача 3

Вычислить НОД двух чисел.

Обозначим НОД двух чисел как НОД(a,b). Тогда справедливо следующее правило:

$$\text{НОД}(a,b)=\begin{cases} a \text{ или } b, \text{ где } a=b; \\ \text{НОД}(a-b,b), \text{ где } a>b; \\ \text{НОД}(a,b-a), \text{ где } a<b. \end{cases}$$

Например:
НОД(36,28) = НОД(36-28,28) = НОД(8,28) = НОД(8,28-8) = НОД(8,20) = = НОД(8,20-8) = НОД(8,12) = НОД(8,12-8) = НОД(8,4) = НОД(8-4,4) = = НОД(4,4) = 4.

```
.8086
.model small
.stack 100h

.data
a     word   36
b     word   90

.code
Start:
    mov    ax,@data
    mov    ds,ax

    mov    ax,a
    mov    bx,b
@next:
    cmp    ax,bx
    je     @exit    ; a = b? тогда выход
    ja     @a       ; a > b? тогда переход
    sub    bx,ax    ; иначе a < b; меняем b
    jmp    @next
@a:
    sub    ax,bx    ; меняем a
    jmp    @next ; повторяем процесс
@exit:

    mov    ah,4Ch
    int    21h

end   Start
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Каким образом реализуется механизм условного выбора в ассемблере?
2. Для каких операций используют команду jmp?
3. Почему при сравнении чисел предварительно необходимо воспользоваться командой сравнения cmp?
4. Каков механизм работы команд j* на основе jbe?
5. Приведите примеры, когда проверка исходного условия требует большего объема ассемблерного кода, чем проверка противоположного.

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Опишите переменную fl размером в байт. В зависимости от значения регистра AX флажку установить значение 1, если AX > 0; 0, если AX = 0; –1 в противном случае.
2. На плоскости задана точка с целочисленными координатами. Проверить, принадлежит ли точка прямоугольнику $[a, b; c, d]$.
3. Написать код, проверяющий на установку флаг ZF. Приведите пример операции (и добавьте ее в код), которая однозначно сбрасывает флаг нуля.
4. Найти наибольшее из трех беззнаковых чисел.
5. Заданы три положительных числа. В переменную treug записать 1, если возможно построить треугольник с указанными длинами сторон.

Практическое занятие № 2.5

ПРЯМАЯ И КОСВЕННАЯ АДРЕСАЦИЯ

1. ЦЕЛИ РАБОТЫ

- Изучить методы оперирования с оперативной памятью.
- Научиться определять адреса переменных и использовать их в косвенной адресации.
- Изучить способы работы с операндами разного размера и оператор уточнения типа.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

На этом занятии рассматриваются вопросы взаимодействия с оперативной памятью. Читателю следует быть особенно внимательным, поскольку указанный вопрос носит фундаментальный характер.

Виды адресации

Одно из преимуществ языка ассемблера – возможность работы с оперативной памятью. Ранее рассматривались особенности адресации памяти в реальном и защищенных режимах работы процессора. Теперь необходимо рассмотреть программные методы доступа к ячейкам оперативной памяти.

Регистровая адресация

При регистровой адресации операнд находится в регистре. Например:

```
mov   ax,bx
push  ds
sub   ah,bh
```

Непосредственная адресация

При непосредственной адресации операнд может быть представлен в виде числа, адреса, кода ASCII, а также иметь символьное обозначение. Например:

```
mov   ax,4C00h           ; операнд 4C00h - число
mov   dl,'M'             ; операнд - код символа 'M'
mov   dx,OFFSET Mes      ; операнд - адрес переменной Mes
five  =  5               ; метке "five" присвоим значение 5
 . . .
mov   cx,five            ; загрузим в регистр значение 5
```

Неявная адресация

При неявной адресации операнд привязан к команде. Например:

```
cbw      ; всегда обрабатывает регистр AL
rep      ; анализирует содержимое регистра CX
```

Непосредственно заданный адрес

Допустим, в оперативную память по определенному адресу требуется записать (либо считать) некоторое значение.

Для указания адреса блока оперативной памяти используются [ ]. Если в скобках указана константа, то такая ссылка называется непосредственной.

Пример:

```
mov   ah,[00A4h]      ; в AH записать байт по адресу 00A4h
mov   bx,[0000h]      ; в BX записать слово по адресу 0000h
ADR   = 0020h
mov   si,[ADR]        ; в SI записать слово по адресу
                      ; ADR (т. е. 0020h)
```

В указанном выше примере компилятор может самостоятельно определить, сколько байт считывается из памяти: на это указывает регистр-приемник.

Для записи в память операнды меняются местами:

```
mov   [00A4h],ah  ; записать байт из AH по адресу 00A4h
```

Непосредственная адресация дает прямой доступ к памяти, но она малоэффективна: адрес переменной устанавливает компилятор, и при перекомпиляциях адреса могут изменяться.

Директива OFFSET

Для определения адреса переменной используется директива OFFSET. Как правило, адрес присваивается допустимому для адресации регистру:

```
mov   регистр, OFFSET переменная
```

Переменная может иметь различный размер и даже структуру. Но в любом случае адресом является первый байт, начиная с которого записываются данные переменной.

Пример:

```
.data
n     BYTE   45
m     DWORD  12345678h
Mass  WORD   1234h, 1000h, 0FFFFh, 0h
 . . .
mov   bx,OFFSET n         ; в BX записывается адрес байта
                          ; со значением 45
mov   si,OFFSET m         ; в SI записывается адрес байта
                          ; со значением 78h
mov   di,OFFSET Mass      ; в DI записывается адрес байта
                          ; со значением 34h
```

Не забывайте, что в реальном режиме адрес всегда задается двухбайтовым операндом! Кроме того, байты записываются в память в обратном порядке (от младших к старшим) по направлению роста адреса.

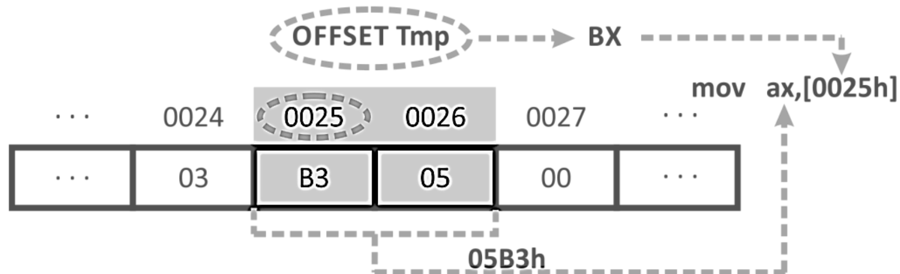
Чтобы лучше понять схему адресации, рассмотрим пример.
Пусть задана переменная Tmp:

```
Tmp   WORD   5B3h
```

Следующий код

```
mov   bx,OFFSET Tmp  ; BX:=адрес переменной
mov   ax,[bx]        ; AX:=слово по адресу,
                     ; хранимому в BX,
                     ; т. е. это само значение переменной
                       Tmp равносилен команде
mov   ax,Tmp         ; AX:=5B3h
```

Если при компиляции переменная Tmp получила адрес 0025h, то схематично работу команд можно представить следующим образом:



Косвенная адресация

В реальном режиме адресации логический адрес определяется парой 16-битных чисел «сегмент:смещение». Поскольку сегмент, как правило, редко требуется изменять самостоятельно (в наших программах об этом заботится система), то адресом будем называть смещение (offset).

В реальном режиме в качестве адресных регистров разрешается использовать регистры SI, DI, BX и BP. Адресация, при которой адрес содержится в одном из этих регистров, называется ***косвенной базовой***.

Примеры:

```
.data
n        word   23h
Mass     byte   10h, 20h, 30h
 . . .
    mov    si,OFFSET n
    mov    ax,[si]        ; альтернатива mov  ax,n
```

```
mov   di,OFFSET Mass
add   ch,[di]      ; CH:=CH+10h
inc   di           ; смещаемся на байт вправо
add   ch,[di]      ; CH:=CH+20h
```

К адресующим регистрам SI, DI, BX, BP можно добавлять числовую константу. Таким образом можно ссылаться на различные байты переменной. Такую адресацию называют ***косвенной со смещением по базе.***

Примеры:

```
.data
n          word   235Ah
Mass       byte   10h, 20h, 30h
  . . .
  mov   si,OFFSET n
  mov   al,[si]        ; AL:=5Ah
  mov   ah,[si+1]      ; AH:=23h

  mov   si,OFFSET Mass
  mov   bl,[si+2]      ; BL:=30h
```

При сложении базовых регистров BX и BP с индексными SI или DI получаем ***косвенную индексную адресацию по базе***. Складывать BX с BP и SI с DI запрещено. Такая адресация удобна при обработке циклических операций с массивами.

Пример:

```
.data
Mass    word   100h, 200h, 300h
  . . .
  mov   bx,OFFSET Mass
  mov   si,0
  mov   ax,[bx+si]     ; AX:=Mass[0] (100h)
  add   bx,2
  mov   cx,[bx+si]     ; AX:=Mass[1] (200h)
  add   bx,2
  mov   cx,[bx+si]     ; AX:=Mass[2] (300h)
  mov   si,3
  mov   ax,[bx+si]     ; AX:= 0002h (захватываем
                       ; части 2-го и 3-го элементов)
```

Комбинация всех приведенных адресаций называется ***косвенной индексной адресацией со смещением по базе***.

Пример:

```
.data
Mass    word   100h, 200h, 300h
  . . .
  mov   bx,OFFSET Mass
  mov   si,2
  mov   ax,[bx+si+2]   ; AX:=Mass[2] (300h)
```

Задача 1

Посчитать количество нулевых байтов в диапазоне 0000..7FFF.

Цикл реализуем с помощью команды условного и безусловного перехода. Таким образом, получим конструкцию цикла с предусловием.

```
.8086
.model small
.stack 100h

.data
kol     word   0

.code
Start:
    mov   ax,@data
    mov   ds,ax

    mov   bx,0h          ; начальный адрес

@next:
    cmp   bx,8000h
    jae   @exit          ; достигли конца? тогда выход
    mov   al,[bx]        ; AL:=байта из памяти
    cmp   al,0           ; сравниваем с нулем
    jne   @notZero       ; AL<>0? - к следующему элементу
    inc   kol            ; иначе увеличиваем счетчик
@notZero:
    Inc   bx
    jmp   @next
```

```
@exit:
    mov   ah,4Ch
    int   21h

end  Start
```

Индексная адресация

Для организации удобной работы с индексом допускается специальная форма записи косвенной ссылки вида

переменная[индекс]

Здесь адрес образуется за счет адреса переменной плюс индекс. Индексом выступают регистры SI, DI, BX или BP, числовые константы или сумма регистров и констант. Подобная форма записи рациональна при обработке массивов.

Пример:

```
Mass    word   100h, 200h, 300h
  . . .
  mov   ax,Mass[0]          ; AX:=0100h
  mov   si,2
  mov   ax,Mass[si]         ; AX:=0200h
  mov   ax,Mass[si+2]       ; AX:=0300h
```

Индексная запись – упрощение косвенной ссылки. В вышеуказанном примере компилятор автоматически возьмет адрес переменной Mass (называется *базой*) и добавит его к индексам в скобках.

Вариативность адресации

Косвенная и индексная записи позволяют осуществлять работу с оперативной памятью несколькими способами. Программист вправе выбирать (и сочетать) любые из перечисленных.

Приведем пример того, сколько вариаций может содержать одна команда:

```
.data
n      BYTE   24h
table  WORD   100h, 200h, 300h
```

Возможности вариаций:

```
mov al,n                mov al,[n]

mov ax,table            mov ax,[table]
```

```
mov ax,table+2           mov ax,[table+2]       mov ax,[table]+2

mov ax,table[si]         mov ax,[table+si]

mov ax,table[bx][di]     mov ax,table[di][bx]
mov ax,table[bx+di]      mov ax,[table+bx+di]
mov ax,[bx+di]+table     mov ax,[bx][di]+table
mov ax,table+[bx][di]
```

Переопределение типа

Для возможности уточнения типа операнда используется команда PTR

```
тип  PTR  значение
```

Эту команду используют в двух случаях:

- размеры операндов не совпадают;
- существует неопределенность, какой размер операнда требуется взять.

Неопределенность может возникнуть при выполнении следующей строки:

```
mov   [bx],12
```

Компилятор не может определить, сколько байт необходимо определить для числа 12. А следующие инструкции корректны:

```
mov   byte PTR [bx],12  ; записать байт по адресу в BX
mov   dword PTR [bx],12 ; записать 4 байта по адресу в BX
```

Оператор PTR также удобен для обращения к отдельным байтам:

```
.data
n     dword  11223344h
Mass  word   5566h, 7788h
  . . .
    mov    al,byte PTR n           ; AL:=44h
    mov    ax,word PTR n           ; AX:=3344h
    mov    ah,byte PTR n+2         ; AH:=22h
    mov    bl,byte PTR Mass[0]     ; BL:=66h
    mov    bh,byte PTR Mass[2]     ; BH:=88h
    mov    dx,word PTR Mass+1      ; DX:=8855h
    mov    word PTR n,9999h        ; n:=11229999h
```

Обратите внимание, что с точки зрения синтаксиса MASM следующие команды идентичны:

```
mov    byte PTR Mass+2,0AAh
mov    byte PTR [Mass+2],0AAh
```

Задача 2

Найти произведение двух беззнаковых чисел размером в слово и записать результат в четырехбайтовую переменную.

В защищенном режиме нет возможности использовать 32-битные регистры. Результат работы операции mul будет храниться в паре регистров DX:AX. Однако нам достаточно знать адрес переменной, и далее по отдельности записать ее младшую часть (из AX) и старшую (из DX).

```
.8086
.model small
.stack 100h

.data
u    word  1000h
v    word  3000h
rez  dword ?

.code
Start:
    mov    ax,@data
    mov    ds,ax

    mov    ax,u
    mul    v                 ; DX:AX:=1000h * 3000h = 300 0000h
    mov    bx,OFFSET rez     ; BX:=адрес искомой переменной
    mov    [bx],ax           ; записываем младшую часть (0000h)
    mov    [bx+2],dx         ; записываем старшую часть (0300h)

    mov    ah,4Ch
    int    21h

end  Start
```

На самом деле код можно уменьшить, вспомнив о возможностях синтаксиса:

```
mov    ax,u
mul    v
mov    word PTR rez,ax           ; записываем младшую часть
mov    word PTR rez+2,dx         ; записываем старшую часть
```

В режиме отладки получим код следующего типа:

```
mov     ax,[0008]
mul     word ptr [000A]
mov     [000C],ax
mov     [000E],dx
```

Очевидно, что DS:0008 и DS:000A – адреса переменных *u* и *v*, а переменная rez расположена по адресу DS:000C. Далее младшее и старшее слово произведения по частям записывается в rez.

```
ds:0000 16 0F 00 B4 4C CD 21 00
ds:0008 00 10 00 30 00 00 00 03
ds:0010 00 00 00 00 00 00 00 00
ds:0018 00 00 00 00 00 00 00 00
```

Операторы расширения типа

Известно, что команды пересылки данных и арифметических операций работают с регистрами одинакового размера, поэтому, к примеру, следующая команда некорректна:

```
add   bx,al   ; ошибка!
```

Эту проблему можно решить, расширив регистр AL до регистра AX. Если в AL беззнаковое число, то достаточно обнулить старший байт:

```
mov   ah,0
add   bx,ax
```

Для положительного числа со знаком действия аналогичны, а для отрицательного в старшем байте приписывают единицы (согласно представлению в дополнительном коде):

```
mov   ah,0FFh
add   bx,ax
```

Однако подобные ситуации можно реализовать и с помощью специальных операторов.

```
cbw
```

Команда расширяет байт до слова со знаком. Операнд находится в регистре AL, результат – в AX.

```
cwd
```

Команда расширяет слово до двойного слова со знаком. Операнд находится в регистре AX, результат – в паре DX:AX.

Пример

```
.data
 N   SBYTE  -101
  . . .
  mov   al,N        ; AL = 9Bh
  cbw               ; AX = FF9Bh
  cwd               ; DX:AX = FFFF FF9Bh
```

Важность косвенной адресации

Ранее были рассмотрены различные способы обращения к переменным. Так, было установлено, что с точки зрения ассемблера команда

```
mov    tmp,ax
```

эквивалентна паре команд

```
mov    bx,OFFSET tmp
mov    [bx],ax
```

Естественным будет вопрос, зачем использовать второй способ, если первый более прост и удобен? Дело в том, что операции с переменными прямо или косвенно используют адресацию. Так, компилятор вместо идентификатора tmp подставляет адрес, а объем байт определяется автоматически по размеру регистра AX.

Не менее важным является следующий факт: *работа с массивами в ассемблере осуществима только с помощью косвенной адресации*.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Перечислите основные виды адресации.
2. Как непосредственно произвести считывание/запись по указанному адресу?
3. Для чего используется директива OFFSET?
4. Что такое косвенная адресация? Перечислите ее возможные вариации.
5. Почему индексная адресация является косвенной?

6. В каких случаях возникает необходимость использования оператора PTR?
7. Какими преимуществами и недостатками обладают операторы расширения типа?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. По адресу 0AAA запишите значение регистра AX, следующие два байта заполните содержимым регистров BL и CH соответственно.
2. Опишите переменную с текстом «Ассемблер». В регистр BL запишите код первого, а в BH – код последнего символа.
3. Напишите программу, вычисляющую количество слов, равных 0 или FFFF, в диапазоне адресов от 0 до 100h. Замечание: подразумевается, что берется по два байта.
4. Задана переменная

```
VeryLong    tbyte   12345678901234567890h
```

Обнулите ее младший байт, самый старший перепишите числом 99h, а байты с 3-го по 6-й задайте значениями 11h. Используйте как можно меньше инструкций.

Практическое занятие № 2.6

ЦИКЛИЧЕСКИЕ ОПЕРАЦИИ. МАССИВЫ. СОРТИРОВКА МАССИВА

1. ЦЕЛИ РАБОТЫ

- Изучить схему работы оператора loop.
- Рассмотреть приемы организации работы с одномерными массивами с использованием косвенной адресации.
- Ознакомиться с дополнительными директивами, позволяющими эффективно обрабатывать массивы.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Циклические операции в ассемблере можно разделить на два типа:

- операции с пост- или предусловием;
- операции со счетчиком.

Оба типа циклов можно реализовать с помощью команд условного и безусловного перехода. Однако циклы со счетчиком допускают специальную реализацию в форме команды loop и ее вариаций.

Оператор цикла loop

loop   метка

Команда loop реализует цикл со счетчиком. Счетчик фиксирован и содержится в регистре CX, хранящем число повторов. За каждый проход значение счетчика уменьшается на единицу. Команда сравнивает CX с нулем и в случае равенства выходит из цикла; в противном случае переходит (по метке) на новый виток.

В общем виде цикл можно представить следующей конструкцией:

```
    mov   cx,n       ; Повторить цикл n раз
метка:
    ;; Тело цикла ;;
    loop  метка      ; Уменьшить CX на 1 и сравнить с нулем
```

При работе с оператором loop необходимо остерегаться следующих ошибок:

- если изначально CX = 0, то в первый раз значение уменьшится на 1 и станет равным CX = FFFF, т. е. цикл повторится максимальное число раз;
- значение счетчика CX нежелательно менять внутри цикла, поскольку это изменит количество витков.

Следует отметить, что диапазон адресов расположения метки цикла ограничен диапазоном –128…127 байт относительно команды loop, т. е. между меткой и loop может находиться в среднем не более 40 – 50 команд размером в 2 – 3 байта каждая.

Задача 1

Вычислить сумму ряда $1 + 2 + \ldots + n$ *при заданном* n.

Поскольку порядок суммирования неважен, то можно учесть особенность работы цикла с регистром CX и суммировать в обратном порядке:

```
.8086
.model small
.stack 100h
```

```
.data
n     word   10h
sum   word   0

.code
Start:
     mov   ax,@data
     mov   ds,ax

     mov   ax,0           ; sum:=0
     mov   cx,n           ; CX:= число повторов
@L:
     add   ax,cx          ; sum:=sum+i
     loop  @L

     mov   sum,ax         ; sum = 88h

     mov   ah,4Ch
     int   21h

end  Start
```

В качестве сумматора здесь использовался регистр AX. Можно уменьшить объем кода, сразу увеличивая переменную sum:

```
     mov   cx,n
@L:
     add   sum,cx
     loop  @L
```

Однако для циклов с большим числом повторов такой код менее эффективен, поскольку операция регистр – регистр быстрее операции память – регистр.

Проанализируем работу цикла в отладчике. Команда loop проверяет равенство CX нулю. Если оно выполняется, то цикл завершается (выполняется следующая команда), иначе, осуществляя переход по адресу CS:0009 и выполнение новой итерации:

```
cs:0005 8B0E0400      mov    cx,[0004]
cs:0009 010E0600      add    [0006],cx
cs:000D►E2FA          loop   0009 ↑
```

При этом автоматически уменьшается значение счетчика-регистра CX

```
cs:0005 8B0E0400        mov     cx,[0004]      cx 000F
cs:0009▸010E0600        add     [0006],cx      dx F798
cs:000D E2FA            loop    0009           si F799
```

По адресу DS:0006 зарезервировано два байта для переменной Sum, в которые и накапливается сумма:

```
ds:0000 4C CD 21 00 10 00 88 00
ds:0008 00 00 00 00 00 00 00 00
ds:0010 00 00 00 00 00 00 00 00
ds:0018 00 00 00 00 00 00 00 00
```

Вариации оператора loop

Операторы loope/loopz и loopne/loopnz помимо проверки равенства счетчика нулю проверяют дополнительные условия:

- loope/loopz – если флаг ZF = 0, то выход из цикла;
- loopne/loopnz – если флаг ZF = 1, то выход из цикла.

Эти циклы часто используют для поиска элемента, неудовлетворяющего / удовлетворяющего заданному условию.

Массивы

При определении переменной как последовательности элементов одного типа можно говорить о задании *одномерного массива* данных. Ранее было показано, что косвенная адресация позволяет обрабатывать массивы удобным образом, обращаясь к элементам массива как по значению, так и по адресу.

Важно понимать, что определенные таким образом переменные записываются в оперативную память последовательно – от первого элемента до последнего.

Любой массив однозначно определяется тремя компонентами:

- тип элементов массива (количество байт на один элемент);
- количество элементов;
- адрес массива (это младший байт, начиная с которого массив записывается в память).

Например, если массив Mass имеет вид

```
Mass    WORD    12AFh, FFFFh, 0A510h, 23h
```

и расположен по адресу 0008, то элементы в памяти будут располагаться по схеме:

Адрес	0008	0009	000A	000B	000C	000D	000E	000F
Значение	AF	12	FF	FF	10	A5	23	00
Индекс (формально)	Mass[0]		Mass[1]		Mass[2]		Mass[3]	
Индекс (фактически)	Mass[0]		Mass[2]		Mass[4]		Mass[6]	

Важно понимать, что индекс элемента массива по факту зависит от типа элементов: формальное обозначение мы подразумеваем, но в ассемблере используем фактическое представление. В указанном примере первый элемент имеет адрес 0008, второй – 000A, третий – 000C, четвертый – 000E.

Полезные директивы

При работе с массивами часто необходимо получить размер массива, количество его элементов и тип каждого элемента. Проделывать вручную это нерационально.

Для упрощения указанных действий в MASM встроены три директивы: TYPE, SIZEOF, LENGTHOF. Как и положено директивам, значения просчитываются на этапе компиляции.

TYPE

Директива TYPE возвращает размер в байтах переменной или элемента массива.

Пример:

```
N      WORD   45                 ; TYPE n = 2
Mass   DWORD  44h, 20h, 400h     ; TYPE Mass = 4
. . .
mov  ax,TYPE n                   ; AX:=2
mov  bx,TYPE Mass                ; BX:=4
```

SIZEOF

Директива SIZEOF возвращает общий размер в байтах переменной или массива.

Пример:

```
n      WORD   45                 ; SIZEOF n = 2
Mass   DWORD  44h, 20h, 400h     ; SIZEOF Mass = 4 * 3 = 12
. . .
mov  ax,SIZEOF n                 ; AX:=2
mov  bx,SIZEOF Mass              ; BX:=12
```

LENGTHOF

Директива LENGTHOF возвращает количество элементов в массиве.

Пример:

```
n      WORD   45                 ; LENGTHOF n = 1
Mass   DWORD  44h, 20h, 400h     ; LENGTHOF Mass = 3
. . .
mov   ax,LENGTHOF n              ; AX:=1
mov   bx,LENGTHOF Mass           ; BX:=3
```

Задача 2

Вычислить сумму элементов массива типа WORD.

```
.8086
.model small
.stack 100h

.data
Mass  word   10h, 20h, 30h, 40h, 50h
sum   word   0h

.code
Start:
     mov   ax,@data
     mov   ds,ax

     mov   ax,0                   ; sum:=0
     mov   cx,LENGTHOF Mass       ; CX:=5
     mov   bx,0                   ; индекс первого элемента
@L:
     add   ax,Mass[bx]            ; AX:=AX+Mass[i]
     inc   bx                     ; следующий индекс
     inc   bx
     loop  @L
     mov   sum,ax
     mov   ah,4Ch
     int   21h

end   Start
```

В этой программе вместо команды

```
add  bx,2
```

используется пара инкрементов

```
inc   bx
inc   bx
```

Это связано с тем, что обе команды занимают два байта памяти и работают быстрее add.

Задача 3

Задан массив чисел размером в байт. Найти максимальный элемент массива.

```
.8086
.model small
.stack 100h

.data
Mass  SBYTE    50h, -20h, 60h, -40h, 40h
max   SBYTE    ?

.code
Start:
     mov   ax,@data
     mov   ds,ax

     mov   bl,Mass[0]     ; допустим, первый – максимальный
     mov   si,1           ; цикл начинаем со второго
     mov   cx,LENGTHOF Mass – 1
@L:
     cmp   Mass[si],bl
     jle   @no            ; если Mass[i]<=макс., идем далее
     mov   bl,Mass[si]    ; иначе – перезаписываем макс.
@no: inc   si             ; индекс следующего элемента
     loop  @L
     mov   max,bl

     mov   ah,4Ch
     int   21h

end   Start
```

Сортировка массива

Одной из наиболее часто возникающих задач при работе с массивами является их сортировка. Прежде всего в этом случае важно время сортировки, которое должно быть минимальным. Скорость сходимости зависит от разных факторов: самого алгоритма, количества элементов и начальной «неупорядоченности» элементов в массиве.

Существуют разные алгоритмы сортировок с различной степенью сложности реализации и скорости работы. Опишем лишь сортировку методом выбора.

Пусть задан массив из N элементов, которые требуется расположить по возрастанию.

1. Считаем, что на i-м шаге элементы с 1-го по (i-1)-й уже отсортированы.
2. Возьмем в качестве минимального i-й элемент.
3. Пробегаем элементы с (i+1)-го по N-й. Если текущий элемент меньше текущего минимального, то запоминаем его индекс.
4. Меняем i-й элемент с минимальным местами.

Можно проследить следующие моменты:

- цикл достаточно повторить N-1 раз, поскольку к этому времени все элементы будут упорядочены;
- менять i-й элемент с минимальным имеет смысл, если их индексы различны (т. е. это не один и тот же элемент).

Для того чтобы проще было строить ассемблерный код, напишем блок сортировки на любом языке высокого уровня, например C++:

```
for(int i = 0; i < n - 1; i++)
{
   int indx = i;

   for(int j = i+1; j < n; j++)
      if(Mass[indx] > Mass[j])
         indx = j;

   if(indx != i)
```

```
    {
      int tmp = Mass[indx];
      Mass[indx] = Mass[i];
      Mass[i] = tmp;
    }
  }
```

Задача 4

Осуществить сортировку выбором для массива типа WORD.

```
.8086
.model small
.stack 100h

.data
Arr   word  40h, 30h, 50h, 30h, 10h

.code
Start:
    mov   ax,@data
    mov   ds,ax

    mov   cx,LENGTHOF Arr-1    ; число витков внешнего цикла
    mov   bx,OFFSET Arr        ; адрес массива
    mov   dx,cx                ; индекс последнего элемента
    shl   dx,1                 ; быстрое умножение на 2
    add   dx,bx
@L:
    mov   di,bx                ; индекс минимального
    mov   si,bx                ; индекс внутреннего цикла
 @next:
    inc   si
    inc   si
    cmp   si,dx                ; пока не перебрали все элементы
    jg    @toswap              ; иначе переход на обмен
    mov   ax,[si]              ; сравниваем текущий с минимальным
```

```
    cmp   ax,[di]
    jge   @next
    mov   di,si             ; запоминаем индекс
    jmp   @next
 @toswap:
    cmp   di,bx
    je    @noswap           ; меняем местами
    mov   ax,[di]
    xchg  ax,[bx]
    mov   [di],ax
 @noswap:
    inc   bx                ; следующий виток
    inc   bx
    loop  @L

    mov   ah,4Ch
    int   21h

end   Start
```

Следует обратить внимание на ряд моментов. Поскольку элементы массива занимают по два байта, то это необходимо учитывать при обработке индексов. В частности, последний элемент имеет индекс 2N-2. В примере показан быстрый способ умножения на 2 с помощью команды SHL сдвига бит влево на один.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. С помощью каких операторов можно организовывать циклы с пред- и постусловием?
2. Опишите работу команды loop.
3. Почему входить в цикл, реализованный с помощью команды loop, рекомендуется только в случае CX неравного нулю?
4. Какие характеристики массива необходимо знать, чтобы корректно осуществлять операции с ним?

5. Докажите «избыточность» дополнительных директив TYPE, SIZEOF, LENGTHOF: любые две совместно позволяют реализовать функцию третьей при ее отсутствии.
6. Какие алгоритмы сортировок массива вам известны?
7. Обоснуйте, почему сдвиг двоичного числа влево на k разрядов равносилен умножению этого числа на 2^k, а вправо – делению на 2^k? В каком случае быстрое умножение/деление приведет к потере данных?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Напишите программу, вычисляющую сумму первых 20 чисел, кратных четырем (начиная с нуля).
2. Вычислите факториал числа при заданном $n \leq 8$.
3. Заполните массив первыми 15 числами последовательности Фибоначчи.
4. Задан массив A. Требуется в массив B записать только четные числа из массива A.
5. Напишите сортировку выбором для массива типа word.
6. Перепишите алгоритм сортировки массива при условии, что его элементы занимают по одному байту памяти.
7. Напишите программу, реализующую пузырьковую сортировку по убыванию элементов.

Практическое занятие № 2.7

СТЕК

1. ЦЕЛЬ РАБОТЫ

- Изучить понятие стека и принципов его работы.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Стек *(англ. stack – стопка) – структура данных, представляющая из себя список элементов, организованных по принципу LIFO (англ. last in – first out, «последним пришёл – первым вышел).*

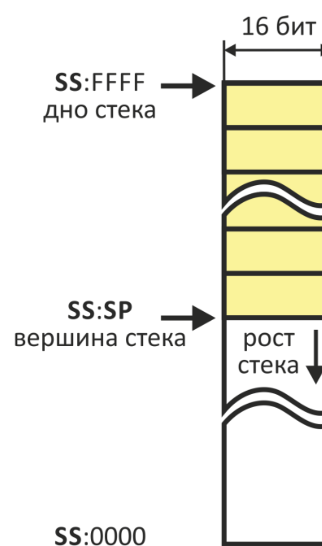
Впервые понятие стека ввёл Алан Тьюринг (1946 г.). Чаще всего принцип работы стека сравнивают со стопкой тарелок: чтобы взять некоторую тарелку из стопки, необходимо предварительно снять все тарелки, лежащие поверх требуемой.

Обычно стек используется для сохранения адресов возврата и передачи аргументов при вызове процедур, также в нём выделяется память для локальных переменных. Полезной функцией стека является временное хранение значений регистров.

Стек в реальном режиме Intel 8086

Схема организации стека в процессоре 8086 следующая:

Стек располагается в оперативной памяти в *сегменте стека* и адресуется относительно сегментного регистра SS (Stack Segment). *Шириной стека* называется размер элементов, которые можно помещать в него или извлекать. В реальном режиме ширина стека равна двум байтам.



Регистр SP (Stack Pointer) – *указатель стека*. Он содержит адрес последнего добавленного элемента. Этот адрес также называется *вершиной стека*. Противоположный конец стека называется *дном*.

Дно стека находится в верхних адресах памяти, т. е. начинает адресацию с FFFF. При добавлении новых элементов в стек значение регистра SP уменьшается, т. е. стек растёт в сторону младших адресов.

Для стека существуют всего две основные операции:

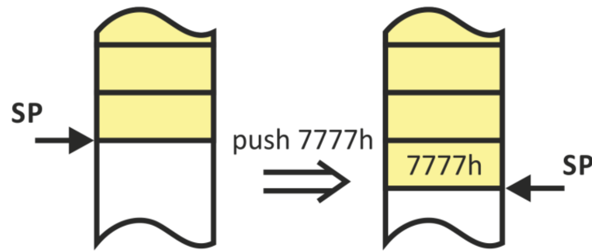
- добавление элемента на вершину стека (PUSH);
- извлечение элемента с вершины стека (POP).

Добавление элемента в стек

push операнд

Операнд может быть 16-битным регистром (в том числе сегментным) или 16-битной переменной в памяти.

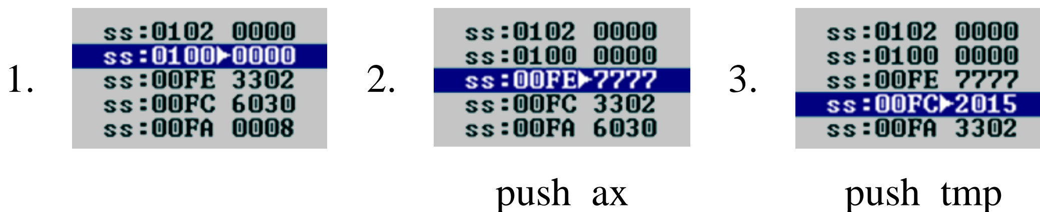
Команда работает следующим образом: значение в регистре SP уменьшается на 2 (так как ширина стека два байта); операнд помещается в память по адресу в SP.



Примеры:

```
push   ax        ; поместить в стек значение регистра AX
push   tmp       ; поместить в стек значение переменной tmp
                 ; (типа WORD / SWORD)
```

В процессе работы видно, как смещается указатель вершины стека



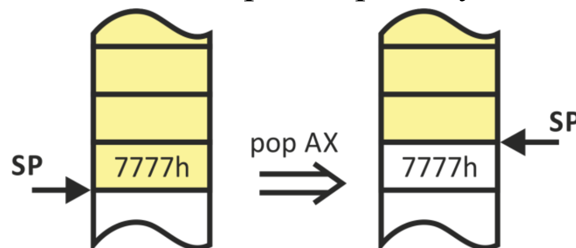
Поскольку директивой .stack 100h был установлен размер стека, то диапазон адресов задается от 0000 до 00FE. Команда push вначале уменьшает значение указателя SP на 2, а затем заталкивает элемент по адресу SS:SP.

Извлечение элемента из стека

```
pop   операнд
```

Операнд может быть 16-битным регистром (в том числе сегментным) или 16-битной переменной в памяти.

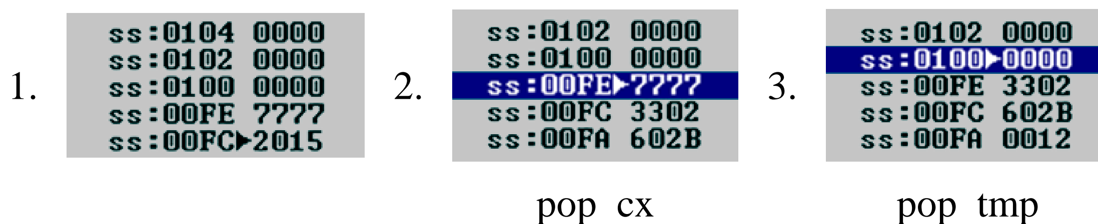
Команда работает следующим образом: операнд читается из памяти по адресу в SP; значение в регистре SP увеличивается на 2.



Примеры:

```
pop   cx         ; записать в CX текущий элемент стека
pop   tmp        ; записать элемент из стека в переменную tmp
                 ; (типа WORD / SWORD)
```

Проанализируем, что происходит при считывании элементов:



pop cx pop tmp

Команда pop вначале извлекает элемент по адресу SS:SP, а затем увеличивает SP на 2.

Команды pusha и popa

pusha

Операция сохраняет в стеке значения регистров общего назначения в следующем порядке: AX, CX, DX, BX, BP, SI, DI.

popa

Операция восстанавливает из стека значения указанных регистров в обратном порядке.

Команды pusha часто применяются в начале процедуры и фрагменте кода, в котором изменяется большое число регистров общего назначения, а popa – в конце процедуры или фрагмента кода. Таким образом производятся сохранение и восстановление значения регистров.

Запись и чтение регистра флагов

pushf

Операция сохраняет в стеке значение регистра флагов.

popf

Операция восстанавливает из стека значение регистра флагов.

Задача 1

Записать в стек значение переменной размером в слово, байт и двойное слово. Восстановить значения из стека в другие переменные соответствующего размера.

Из трех чисел напрямую в стек можно переслать только число размером в слово. Байтовое число требуется расширить до двух байт, а четырехбайтовое «запаковать» по частям. При считывании из стека операции будут обратными

```
.8086
.model small
.stack 100h

.data
n      BYTE   12h
m      WORD   1234h
k      DWORD  12345678h
n1     BYTE   ?
m1     WORD   ?
k1     DWORD  ?

.code
Start:
    mov   ax,@data
    mov   ds,ax

    mov   al,n
    cbw                        ; расширяем AL до AX
    push  ax                   ; записать n (0012)
    push  m                    ; записать m (1234)
    push  WORD PTR k           ; записать младшее слово k (5678)
    push  WORD PTR k+2         ; записать старше слово k (1234)

    pop   WORD PTR k1+2        ; восстановление в обратном
    pop   WORD PTR k1          ; порядке
    pop   m1
    pop   ax
    mov   n1,al

    mov   ah,4Ch
    int   21h

end  Start
```

Стек позволяет реализовывать некоторые типы операций на простом уровне.

Задача 2

Дана строка. С помощью стека инвертировать ее и записать в другую переменную.

```
.8086
.model small
.stack 100h
.data
StrA   BYTE  "Hello"
Len    =     LENGTHOF StrA; Число символов в строке
StrB   BYTE  Len  DUP(?)    ; Резервируем память для
                            ; новой строки
.code
Start:
    mov   ax,@data
    mov   ds,ax

    mov   cx,Len
    mov   si,0
@L1:
    mov   al,StrA[si]       ; Считываем ASCII код символа
    cbw                     ; и расширяем его до двух байт
    push  ax                ; Записываем его в стек
    inc   si
    loop  @L1

    mov   cx,Len
    mov   si,0
@L2:
    pop   ax                ; Считываем из стека код символа
    mov   StrB[si],al       ; и записываем его в StrB
    inc   si
    loop  @L2

    mov   ah,4Ch
    int   21h
end   Start
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что такое стек? Опишите механизм его работы.
2. Какие регистры обеспечивают работу стека?
3. Какие логические ошибки можно допустить, используя стек для временного хранения значения регистров?
4. Приведите пример ситуаций, в которых требуется сохранить значение регистра флагов для последующего восстановления.
5. Покажите, что функционал команд pusha/popa можно полностью реализовать без их присутствия.

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Напишите программу, осуществляющую обмен двух переменных значениями через стек.
2. Предложите альтернативный вариант инициализации регистра DS с помощью стека, который обычно осуществляется строками
   ```
   mov    ax,@data
   mov    ds,ax
   ```
3. Напишите программу, которая «запаковывает» в стек число типа QWORD.
4. Как с помощью стека организовать несколько вложений цикла loop?

Практическое занятие № 2.8

ПРЕРЫВАНИЯ

1. ЦЕЛЬ РАБОТЫ

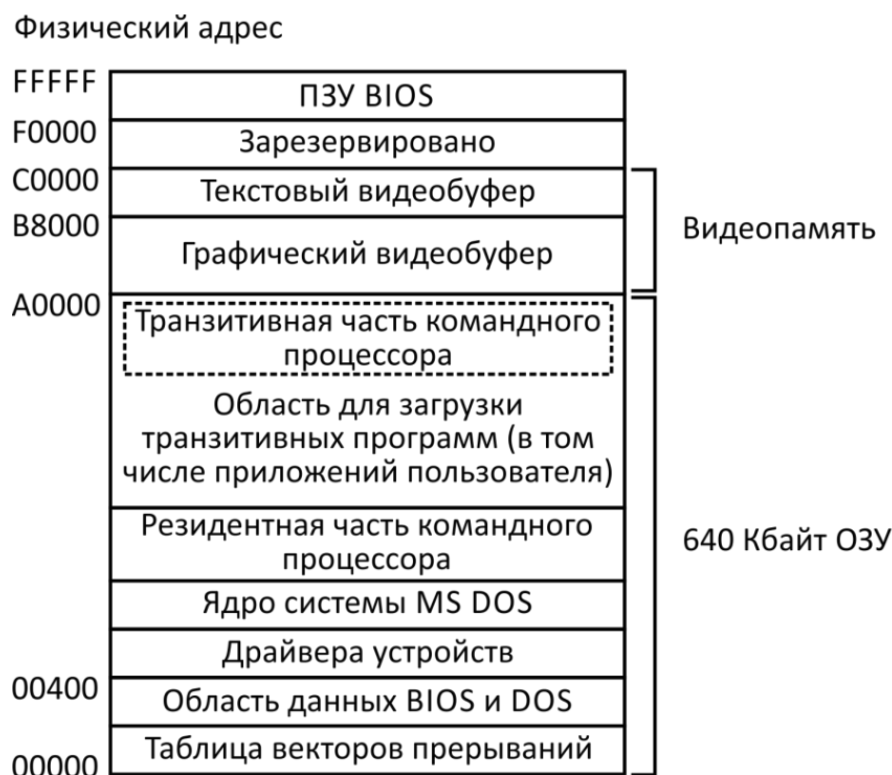
- Изучить механизм обработки внешних и внутренних процессов на уровне прерываний.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Организация памяти в реальном режиме

В реальном режиме адресации первые 640 Кбайт адресного пространства используются в качестве ОЗУ операционной системой и

прикладными программами. Следующий участок зарезервирован под видеопамять. Зарезервированный участок памяти может использоваться платами расширения и контроллерами устройств. Верхний блок памяти предназначен для размещения системного ПЗУ. Более подробно это можно представить в виде схемы



Прерывания

Прерывание *(interrupt) – сигнал, сообщающий процессору о наступлении какого-либо события.*

При возникновении прерывания выполнение текущей последовательности команд приостанавливается и управление передаётся *обработчику прерывания*, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

В зависимости от источника возникновения сигнала прерывания делятся:

- на *асинхронные*, или *внешние* (аппаратные) – события, которые исходят от внешних источников (например, периферийных устройств) и могут произойти в любой произвольный

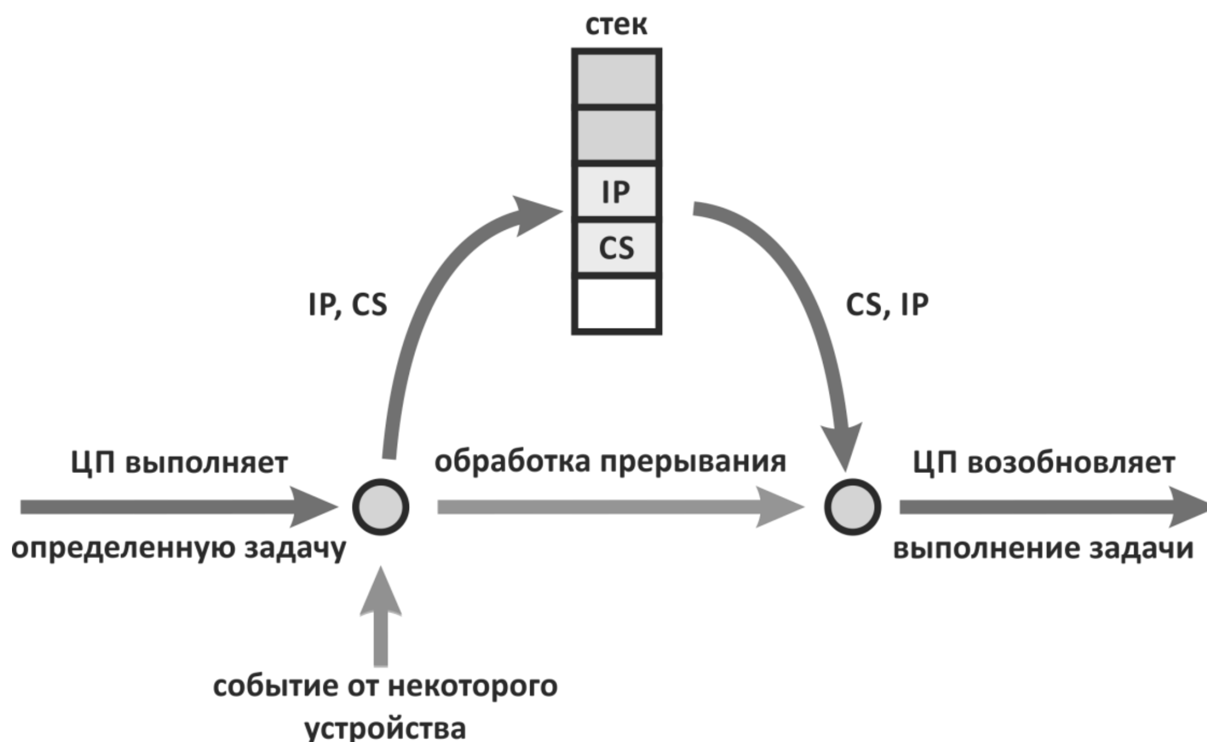
момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши;

- *синхронные*, или *внутренние* – события в самом процессоре как результат нарушения каких-либо условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти или недопустимый код операции;
- *программные* (частный случай внутреннего прерывания) – инициируются исполнением специальной инструкции в коде программы, т. е. генерируются искусственно; как правило, используются для обращения к функциям встроенного ПО, драйверов и операционной системы.

Схема работы прерывания

Предположим, что в ходе выполнения центральным процессором программы в некотором внешнем устройстве произошло событие (например, нажатие клавиши), на которое операционная система должна немедленно отреагировать. В этом случае устройство посылает в центральный процессор специальный сигнал, называемый *сигналом прерывания,* или просто *прерыванием*. Получив такой сигнал, центральный процессор прерывает выполнение текущей программы и передает управление операционной системе, определяющей, какое событие произошло, и соответствующим образом реагирует на это событие. Когда обработка прерывания операционной системой закончена, центральный процессор возобновляет выполнение прерванной программы с той команды, на которой работа программы оборвалась. Адрес команды, которая должна быть следующей при выполнении программы центральным процессором, хранится в паре регистров CS:IP. Когда в центральный процессор приходит сигнал прерывания, то перед обработкой прерывания, для того чтобы впоследствии работа программы была возобновлена корректно, происходит сохранение в стеке текущего значения регистра флагов и адреса следующей команды.

Обработку прерывания можно представить следующей схемой:



Команда int

int   номер_прерывания

Команда вызывает процедуру обработки прерывания, помещая перед этим в стек состояние регистра флагов.

Перед выполнением команды в определенные регистры необходимо загрузить значения соответствующих параметров. Иными словами, одному прерыванию могут соответствовать различные функции.

Функцию прерывания принято называть по содержимому регистра АН. Полные списки прерываний и функций можно найти в специальных справочниках.

Среди наиболее распространенных прерываний следующие:

- int   10h – видеослужбы;
- int   16h – работа с клавиатурой;
- int   17h – работа с принтером;
- int   1Ah – работа с временем;
- int   1Ch – прерывание по таймеру;
- int   21h – функции MS DOS.

Ограничим изучение возможностей прерываний на int 21h. Тем более, что одну из функций этого прерывания мы использовали во всех программах этой части.

Прерывание int 21h

Функция 4Ch

Регистр AH	Операция	Входные параметры	Выходные параметры
4Ch	Завершение текущей программы (процесса)	AL = код выхода (необязателен)	Нет

Функция 01h

Регистр AH	Операция	Входные параметры	Выходные параметры
01h	Ввод символа с эхом	Нет	AL := код введенного символа

Функция 02h

Регистр AH	Операция	Входные параметры	Выходные параметры
02h	Вывод символа на экран	DL = коду выводимого символа	Нет

Функция 08h

Регистр AH	Операция	Входные параметры	Выходные параметры
08h	Ввод символа без эха	Нет	AL := код введенного символа

Функция 09h

Регистр AH	Операция	Входные параметры	Выходные параметры
09h	Выводит строку на экран. Строка должна завершаться символом «$»	DS:DX = адрес строки (чаще достаточно DX, поскольку DS определяется автоматически)	Нет

Эхо подразумевает, что вводимый символ отображается на экране.

Задача 1

Осуществить ввод символа с эхом, далее вывести этот символ на экран. Консоль должна закрыться по нажатию любой клавиши.

```
.8086
.model small
.stack 100h

.code
Start:
    mov   ax,@data
    mov   ds,ax

    mov   ah,01h        ; Функция ввода символа с эхом
    int   21h           ; После выполнения в AL запишется
                        ; код нажатого символа
    mov   ah,02h        ; Функция вывода символа по коду
    mov   dl,al         ; Выведем тот же символ, что и ввели
    int   21h

    mov   ah,08h        ; Функция ввода без эха
    int   21h           ; ЦП ждет нажатия любой клавиши

    mov   ah,4Ch        ; Функция завершения процесса
    int   21h

end   Start
```

Задача 2

Вывести сообщение «Hello, assembler!».

```
.8086
.model small
.stack 100h

.data
Mess  BYTE  "Hello, assembler!$"
```

```
.code
Start:
      mov   ax,@data
      mov   ds,ax

      mov   ah,09h                ; функция вывода строки
      mov   dx,OFFSET Mess        ; DX:= адрес строки
      int   21h

      mov   ah,08h                ; задержка консоли
      int   21h

      mov   ah,4Ch
      int   21h

end   Start
```

Задача 3

С помощью функции 01h организовать ввод текста в переменную. При нажатии на Enter ввод завершается.

```
.8086
.model small
.stack 100h

.data?
text  byte  100  dup(?)           ; резерв на 100 символов

.code
Start:
      mov   ax,@data
      mov   ds,ax

      mov   bx,OFFSET text        ; BX:= адрес переменной
      mov   ah,01h                ; задаем функцию
@next:
      int   21h                   ; вызов прерывания
      cmp   al,0Dh                ; проверяем нажатие Enter
```

```
    je      @end            ; если нажали Enter, то выход
    mov  [bx],al            ; иначе добавляем символ
    inc     bx
    jmp   @next
@end:
    mov  ah,4Ch
    int     21h

end   Start
```

Задача 4

Осуществить вывод положительного числа размером в слово.

Для вывода числа потребуется перевести его в строку. Вначале необходимо получить все цифры числа делением на 10 и взятием остатка (т. е. цифры). Однако цифры будут получены в обратном порядке. Проблему решает стек: если записывать каждую цифру в стек, то при считывании получим цифры в правильном порядке.

Чтобы избежать проблемы с делением (неполному частному может не хватить места в AL), осуществляем его через пару DX:AX. На каждом новом шаге DX необходимо сбрасывать, поскольку при делении в него записывается цифра.

Последнее, что потребуется – вывести в цикле символы из стека. Однако цифры необходимо скорректировать до соответствующих им символов в таблице ASCII. Так как цифры кодируются в диапазоне от 30h до 39h, то к цифре просто добавим 30h.

```
.8086
.model small
.stack 100h

.data
n     word  12345

.code
Start:
    mov   ax,@data
    mov   ds,ax
```

```
    mov   ax,n      ; AX:= число
    mov   cx,1      ; CX:= счетчик цифр
    mov   bx,10     ; требуется для деления

@next_sim:
    mov   dx,0      ; сбрасываем старшую часть
    cmp   ax,bx     ; сравниваем с 10
    jb    @end      ; если меньше, то завершаем
    div   bx        ; иначе делим число на 10
    push  dx        ; записываем цифру в стек
    inc   cx        ; увеличиваем счетчик цифр
    jmp   @next_sim
@end:
    push  ax        ; хотя бы одна цифра есть всегда
    mov   ah,02h    ; задаем функцию вывода символа
@L:
    pop   dx        ; извлекаем цифру
    add   dl,30h    ; скорректируем ее до ASCII кода
    int   21h       ; выводим
    loop  @L

    mov   ah,08h
    int   21h

    mov   ah,4Ch
    int   21h

end  Start
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что представляет собой прерывание и в каких случаях оно может возникнуть?
2. Опишите схему обработки прерывания процессором.
3. Какие прерывания вам известны?
4. Какие основные рассмотренные в работе функции прерывания 21h вам известны?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Доработайте программу ввода текста так, чтобы после ввода в переменную text дописывался символ конца строки и она выводилась на экран.
2. Напишите программу вывода чисел, способную выводить как положительные, так и отрицательные числа.

Практическое занятие № 2.9

ПРОЦЕДУРЫ

1. ЦЕЛИ РАБОТЫ

- Изучить методику логической структуризации ассемблерной программы с помощью процедур.
- Рассмотреть способы вызова процедур и различие между параметрами значениями и параметрами ссылками.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Часто большую задачу удобно разбить на ряд более мелких подзадач. Сначала описывается общая архитектура приложения, а затем реализуется код подзадач. В программировании такой подход называют *проектированием сверху-вниз*. Подзадачи реализуются в форме подпрограмм, называемых *функциями*, *процедурами* или *методами*.

Процедуры в ассемблере

Понятия функции или метода свойственны более современным языкам программирования, в частности объектно-ориентированным. Ассемблер появился задолго до новых парадигм программирования, поэтому понятие подзадачи связывают с термином процедуры.

Процедура *(в ассемблере) – именованный блок команд, оканчивающийся оператором возврата.*

Для описания процедуры используются директивы PROC и ENDP, а также команда возврата ret:

```
имя_процедуры  PROC
. . .
ret
имя_процедуры  ENDP
```

К особому виду процедур относится стартовая, она способна заменить конструкцию Start: – end Start. Для стартовой процедуры команда ret не нужна.

Для вызова процедуры используется команда call:

```
call   имя_процедуры
```

Задача 1

Создать стартовую процедуру с именем Main и в ней вызвать дополнительную процедуру MyProc, суммирующую значения основных регистров в AX.

```
.8086
.model small
.stack 100h

.code
; Основная (стартовая) процедура
Main   PROC
       call  MyProc
       mov   ah,4Ch
       int   21h
Main   ENDP

; Дополнительная процедура
MyProc  PROC
        add   ax,bx
        add   ax,cx
        add   ax,dx
        ret
MyProc  ENDP

end    Main
```

Отладчик выдает следующий код:

```
cs:0000►E80400         call    0007
cs:0003 B44C           mov     ah,4C
cs:0005 CD21           int     21
cs:0007 03C3           add     ax,bx
cs:0009 03C1           add     ax,cx
cs:000B 03C2           add     ax,dx
cs:000D C3             ret
```

При вызове процедуры осуществится переход в соответствующую секцию кода, а значение регистра IP запишется в стек

```
ss:0106 601B
ss:0104 001F
ss:0102 2492
ss:0100 602C
ss:00FE 0003
```

При выполнении команды ret значение указателя IP восстанавливается и программа продолжает выполнение основного кода.

Механизм вызова процедуры

При выполнении команды call в стек записывается значение счетчика команд (адрес следующей команды). После этого процессор переходит по указанному для кода процедуры адресу и выполняет его. По достижению команды ret в регистр IP из стека возвращается сохраненное значение счетчика команд и программа продолжает свою работу.

Такой механизм вызова и обработки процедур позволяет создавать многоуровневые вложения процедур.

Метки перехода

По умолчанию MASM рассматривает метки как *локальные*, т. е. доступные только внутри текущей процедуры. Поэтому компилятор выдаст ошибку, если вы попытаетесь перейти по метке из одной процедуры в другую. Такой подход логичен: процедура является вполне замкнутой логической сущностью, поэтому «прыгать» из одной процедуры в другую по метке не рекомендуется.

Однако (пусть и редко) такая необходимость возникает. Тогда можно воспользоваться *глобальной меткой*, которая завершается двумя двоеточиями:

global_label::

К такой метке можно перейти из любого места программы.

Регистры как параметры процедуры

Для расширения возможностей процедуры зачастую требуется передавать ей *параметры*. В качестве носителей параметров можно использовать регистры.

Задача 2

Написать процедуру, суммирующую элементы массива типа WORD.

Процедуре будут передаваться два параметра: в CX – количество элементов, в SI – адрес массива. Результат суммы сохраняется в AX:

```
.8086
.model small
.stack 100h

.data
Mass   WORD   10h, 20h, 30h
Sum    WORD   ?

.code
Main   PROC
       mov   ax,@data
       mov   ds,ax

       mov   cx,LENGTHOF Mass        ; CX:=количество элементов
       mov   si,OFFSET Mas           ; SI:=адрес массива
       call   SumProc

       mov   Sum,ax                  ; результат суммы – в AX

       mov   ah,4Ch
       int    21h
Main   ENDP

SumProc  PROC
           mov   ax,0
```

```
@next:
        add     ax,[si]
        inc     si
        inc     si
        loop    @next
        ret
SumProc ENDP

end     Main
```

Тонкости работы с параметрами. Команда USES

При передаче параметров через регистры необходимо учитывать, что, как правило, значения определенных регистров меняются. Это может быть крайне нежелательно. Яркий пример – предыдущая программа.

Решить эту проблему можно несколькими способами.

1. В самом начале процедуры скопировать значения регистров CX и SI в другие регистры (например, DX и DI), а перед выходом из процедуры – восстановить их обратно. Этот способ малоэффективен, поскольку другие регистры также могут быть заняты.
2. Воспользоваться стеком для временного хранения. Это наиболее универсальный способ.
3. Сохранить значения регистров в дополнительных переменных. Способ достаточно универсален, но требует описания переменных.

Есть более короткое решение – оператор USES. В начале процедуры он записывает указанные регистры в стек, а перед завершением их восстанавливает:

```
SumProc PROC  USES  cx si
    . . .
SumProc ENDP
```

На самом деле оператор USES cx si при компиляции просто автоматически дописывает в начале и в конце процедуры следующий код:

```
push   cx
push   si
   . . .
pop    si
pop    cx
```

Обратите внимание, что регистр AX не указывается в USES, поскольку он возвращает результат работы процедуры.

Переменные как параметры процедуры

Ранее мы рассмотрели *регистровый* способ передачи параметров процедуре. Однако в общем случае процедура может содержать параметры, хранящиеся в переменных.

Директива описания процедуры в полной форме имеет следующий синтаксис:

```
имя_процедуры  PROC  параметр_1, параметр_2, ...,
параметр_N
   ; тело процедуры
      ret
имя_процедуры  ENDP
```

Каждый параметр определяется именем и типом:

имя_параметра:тип

В таком случае аргументы передаются через стек (*стековый* способ). Поэтому перед вызовом процедуры их нужно записать в стек.

Компилятору необходимо указать, в каком порядке будут передаваться аргументы в стек. Это называется соглашением вызова[4]. Обычно используют стандартное соглашение stdcall, оно указывается в директиве .model:

```
.model   small, stdcall
```

Применяя соглашение stdcall, нужно помнить, что аргументы необходимо записывать в стек в обратном порядке!

---

[4] Существует несколько соглашений вызова с разными порядками записи аргументов в стек: stdcall, pascal, register, cdecl.

Задача 3

Написать процедуру, суммирующую три числа типа WORD. Результат записать в AX.

```
.8086
.model small,stdcall
.stack 100h

.data
num1    word  10h
num2    word  20h
num3    word  30h

.code
Main    PROC
        mov   ax,@data
        mov   ds,ax

        push  num3
        push  num2
        push  num1
        call  Sum3        ; после выполнения сумма запишется в AX

        mov   ah,4Ch
        int   21h
Main    ENDP

Sum3    PROC  n1:word, n2:word, n3:word
        mov   ax,n1
        add   ax,n2
        add   ax,n3
        ret
Sum3    ENDP

end     Main
```

Локальные переменные

Внутри процедуры можно описывать временные, т. е. *локальные переменные*. Они содержатся в стеке и существуют только во время выполнения процедуры.

Описание локальной переменной начинается с директивы LOCAL:

```
LOCAL  имя_переменной:тип
```

Локальная переменная видна только внутри процедуры.

Директива INVOKE

Для более удобного и гибкого вызова процедур с параметрами была разработана специальная директива INVOKE, альтернативная команде call:

```
INVOKE  имя_процедуры [, список_аргументов]
```

Такой синтаксис приближает разработчика к уже привычному вызову процедуры (функции) в языках высокого уровня. Теперь аргументы не нужно заносить в стек вручную: их достаточно перечислить при вызове процедуры, причем в обычном порядке.

Так, в задаче 3 следующий вызов функции Sum3

```
push  num3
push  num2
push  num1
call  Sum3
```

можно заменить одной командой

```
INVOKE  Sum3, num1, num2, num3
```

Однако в таком случае для функции необходимо описать ее прототип.

Прототип функции

Если вам доводилось работать с языком C (C++), то наверняка при работе с процедурами/функциями вы сталкивались с необходимостью описывать их прототипы.

Прототип функции *– объявление функции, содержащее ее заголовок.*

Как правило, прототипы описываются в начале программы, перед сегментом данных .data. Синтаксис следующий:

```
имя_процедуры  PROTO [параметр_1]:тип,
                     [параметр_2]:тип,
                     ...,
                     [параметр_N]:тип
```

Прототипы необходимы, если вызов процедуры производится выше, чем она описана, и обязательны, если используется директива INVOKE.

Задача 4

Перепишите задачу 3, используя директиву INVOKE.

```
.8086
.model small,stdcall
.stack 100h

Sum3   PROTO  :word, :word, :word      ; имена переменных в
                                       ; прототипах необязательны
.data
num1   word  10h
num2   word  20h
num3   word  30h

.code
Main   PROC
        mov   ax,@data
        mov   ds,ax

      INVOKE  Sum3, num1, num2, num3

      mov   ah,4Ch
      int     21h
Main   ENDP

Sum3   PROC  n1:word, n2:word, n3:word
         mov   ax,n1
         add   ax,n2
         add   ax,n3
         ret
Sum3   ENDP

end    Main
```

Прототипы необходимы компилятору для сверки типов аргументов и проверки корректности описания заголовка в целом.

Есть еще одна важная функция прототипов. Их можно сосредоточить в отдельном (заголовочном) файле, а реализацию процедур выносить в другие файлы. Последние можно скомпилировать в *динамически подключаемую библиотеку* (.dll) процедур[5]. Это позволит вызывать их из любой программы. В свою очередь, файл с прототипами – «меню» из процедур и функций, которые можно подключить в программе. Похожий принцип используется в C/C++.

Параметры-значения и параметры-указатели

Пусть требуется описать процедуру обмена двух переменных значениями. Предположим, что создана процедура Swap с двумя параметрами и корректным кодом, меняющим значение переменных; также описан ее прототип.

Однако, вызвав процедуру командой

```
INVOKE  Swap, A, B,
```

мы обнаружим, что обмена не произошло!

Проблема кроется в способе передачи параметров процедуре и их дальнейшей обработке. В нашем случае в стек изначально помещаются значения переменных A и B. Процедура считывает значения этих параметров и записывает их в локальные параметры (те, что указаны в ее заголовке), т. е. на самом деле все дальнейшие операции производятся не с переменными A и B, а с их копиями!

Чтобы процедура получила прямой доступ к переменной, необходимо передавать адрес этой переменной. Часто говорят, что это *параметр-указатель*.

Указатель* (*pointer*) *– переменная, хранящая адрес другой переменной.

---

[5] Примеры создания динамически подключаемой библиотеки будут приведены в разделах, посвященных основам программирования с использованием Windows API.

Параметры-указатели требуется описывать с оператором PTR, выполняющим в этом случае роль указателя на определенный тип.

Задача 5

Напишите процедуру обмена двух чисел.

```
.8086
.model small,stdcall
.stack 100h

Swap    PROTO :PTR word, :PTR word

.data
A       word  10h
B       word  20h

.code
Main    PROC
        mov   ax,@data
        mov   ds,ax

     ; передаем указатели на переменные
     INVOKE  Swap, OFFSET A, OFFSET B

        mov   ah,4Ch
        int   21h
Main    ENDP

Swap    PROC  pA:PTR word, pB:PTR word
        mov   si,pA              ; указатель на A
        mov   di,pB              ; указатель на B
        mov   ax,[si]
        xchg  ax,[di]
        mov   [si],ax
        ret
Swap    ENDP

end     Main
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Для каких целей используются процедуры?
2. Опишите синтаксис вызова процедуры без параметров.
3. Опишите механизм вызова и обработки процедуры.
4. Каким образом работает механизм передачи параметров процедуре через регистры?
5. Что такое соглашение вызова? Объясните, почему для соглашения stdcall организован обратный порядок передачи параметров?
6. Какова область видимости локальных переменных процедуры?
7. Для чего предназначена директива INVOKE? Что такое прототип процедуры?
8. Чем отличаются параметры-значения от параметров-указателей? Объясните различия на основе механизма обработки параметров.

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Написать процедуру Print, выводящую на консоль указанную строку. Параметром является адрес строки, который передается через DX.
2. Напишите функцию, возвращающую в AX индекс первого четного числа заданного массива. Параметры передаются через стек. Реализуйте код
   - с командой call;
   - с директивой INVOKE.
3. Напишите программу обмена двух переменных значениями с использованием и без использования процедуры. С помощью отладчика проследите, какие операции происходят при вызове процедуры. Почему в программах, требующих максимальной скорости выполнения, процедур следует избегать?
4. Исследуйте с помощью отладчика механизм вызова процедуры с параметрами, передаваемыми через стек. Постарайтесь объяснить, какую функцию при этом выполняют регистры SP и BP.

Практическое занятие № 2.10
МАКРОСЫ

1. ЦЕЛИ РАБОТЫ

- Изучить функциональные возможности макроопределений.
- Расширить представление о возможностях синтаксиса ассемблера.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Часто возникает необходимость неоднократного повтора определенного участка кода. Чтобы не загромождать программный код копиями команд, используются макросы.

Макрос *– символьное имя, заменяющее несколько команд ассемблера.*

Макрос похож на процедуру, однако:

- в отличие от процедуры – это именованный блок кода;
- аргументы макроса не записываются в стек.

Макросы полезны, когда в программе часто повторяются одинаковые участки кода: прерывания на ввод/вывод символов и строк, операции записи и извлечения нескольких регистров из стека и т. д. Конечно, их можно оформить как процедуры, однако последние увеличивают количество операций, что, например, в циклических конструкциях может привести к дополнительным издержкам по времени выполнения.

Макросы можно оформлять внутри самой программы либо в отдельном файле, если предполагается использовать макросы в различных программах. Файлы с макросами имеют расширение .inc (include – подключить) или .asm.

Макрос описывается следующим образом:

```
  имя_макроса  MACRO  [параметры]
    ;; код макроса
ENDM
```

Для вызова макроса указывают его имя (и параметры). Компилятор автоматически подставляет вместо имени необходимый код.

Для подключения пакета макросов используется директива INCLUDE. Если файл с макросами находится в другой директории, то требуется указать полный путь к файлу.

Задача 1

Сформируйте файл с макросами сохранения адреса сегмента данных, задержкой консоли и завершением программы. Продемонстрируйте их работу.

Для начала создадим файл с макросами MyMacros.inc:

```
;; Сохранение адреса сегмента данных ;;
DataAdress MACRO
                mov   ax,@data
                mov   ds,ax
ENDM

;; Завершение программы ;;
Exit MACRO
      mov   ah,4Ch
      int     21h
ENDM

;; Ожидание нажатия клавиши ;;
PressKey  MACRO
              mov   ah,08h
              int     21h
ENDM
```

Теперь файл можно подключить в основной программе:

```
.8086
.model small,stdcall
.stack 100h

INCLUDE  MyMacros.inc    ; Подключаем макросы

.code

Main   proc
     DataAdress              ; DS:=адрес сегмента данных
     PressKey                ; Ждем нажатия клавиши
     Exit                    ; Выход
Main   endp

end    Main
```

Код отладчика:

```
mov     ax,602B
mov     ds,ax
mov     ah,08
int     21
mov     ah,4C
int     21
```

Очевидно, что использование макросов уменьшило объем кода основной программы, а читабельность при этом даже улучшилась.

Макросы с параметрами

Макросам можно передавать аргументы. Формальные параметры выполняют роль носителей аргумента и по существу лишь скрывают за собой фактический операнд. В отличие от процедур за сохранность значений регистров в макросах полную ответственность несет программист.

Задача 2

Добавьте к списку макросов макрос на вывод строки.

Допишем в MyMacros.inc следующий код:

```
NEWLINE    EQU    13, 10   ; Переход на новую строку

;; Печать строки ;;
Print MACRO Text
        push  ax                    ; Сохраняем значения AX и DX
        push  dx
        mov   ah,09h
        mov   dx,OFFSET Text
        int   21h
        pop   dx                    ; Восстанавливаем AX и DX
        pop   ax
ENDM
```

Код основной программы:

```
.8086
.model small,stdcall
.stack 100h

INCLUDE  MyMacros.inc     ; Подключаем макросы
```

```
.data
Mess   byte  "Hello, aasembler!", NEWLINE, "$"

.code
Main   proc
        DataAdress       ; DS:=адрес сегмента данных
        Print   Mess     ; макрос с параметром
        PressKey         ; Ждем нажатия клавиши
        Exit             ; Выход
Main   endp

end    Main
```

Идентификатор NEWLINE играет роль константы: везде, где он присутствует, будет подставлено два байта со значением 13 и 10, что отвечает кодам нажатия клавиши Enter и переводу курсора в начало строки.

Макросы в определенном смысле имеют больше возможностей, чем процедуры. Не случайно они упоминаются даже в названии ассемблера – Macro Assembler. Мы рассмотрели только простейшие возможности описания макросов. В следующей части книги читатель убедится, что макросы – великолепный инструмент, преображающий синтаксис языка ассемблер.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что такое макрос в ассемблере?
2. Чем отличаются механизмы работы макроса от процедуры?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Напишите макрос, выводящий строку из звездочек.
2. Создайте макрос Input с двумя параметрами, обрабатывающий ввод текстовой строки в переменную (не более 100 символов). Первый параметр – указатель на переменную, в которую считывается текст; второй – регистр CX, в который записывается число введенных символов.

Часть 3. MACRO ASSEMBLER: IA-32. ТЕХНОЛОГИЯ WINDOWS API

Практическое занятие № 3.1

АРХИТЕКТУРА IA-32. ПЕРЕХОД НА 32-БИТНОЕ ПРОГРАММИРОВАНИЕ

1. ЦЕЛИ РАБОТЫ

- Изучить основные особенности 32-битной архитектуры Intel x86.
- Переориентировать ранее полученные знания о 16-битной архитектуре на 32-битную.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Современные приложения достаточно сложным образом взаимодействуют с операционной системой. Это требует более трудоемкой организации программы. Поэтому ниша ассемблера в разработке подобных проектов вытиснилась такими языками, как C++, C#, Delphi и т. д.

Однако за последнее время была проделана немалая работа, расширившая возможности классического ассемблера. Это позволяет писать весьма сложные приложения с простотой, присущей языкам высокого уровня.

Процессор i386

IA-32 (Intel Architecture, 32-bit) *– третье поколение архитектуры x86, ознаменовавшееся переходом на 32-разрядные вычисления.*

Intel 80386 (i386) *– первый 32-битный x86-совместимый процессор третьего поколения фирмы Intel (17.10.1985).*

Переход на 32-битную архитектуру позволил сделать защищенный режим работы процессора основным, что открыло широкие возможности программистам.

В отличие от реального режима со специфической сегментацией памяти защищенный режим позволяет рассматривать адресное пространство в линейной форме. Поскольку адрес задается 32-битным числом, то всего возможно адресовать $2^{32}\text{байт} = 4\text{Гбайта}$ памяти в рамках приложения. При этом диапазон изменения адресов варьируется от 0000000h до FFFFFFFh. Схематично это можно представить так:

Адрес	00000000	00000001	00000002	. . .	FFFFFFFE	FFFFFFFF
Байт	78	00	5D	. . .	00	0F

Защищенный режим серьезно упрощает работу, поскольку множество рутинных и сложных задач переложено на операционную систему.

В IA-32 сегментные регистры выполняют иную роль и являются указателями на *дескрипторы сегмента* (segment descriptor). Дескрипторы хранятся в специальных системных *таблицах дескрипторов* (descriptor table). Как правило, на программу выделяется три секции: кода, данных и стека (за них отвечают регистры CS, DS и SS соответственно).

В защищенном режиме можно выделить несколько моделей сегментации памяти:

- *линейно-сегментная*. Дескрипторы всех сегментов указывают на один участок памяти. Сегмент задается 64-битным числом-дескриптором, которое хранится в таблице глобальных дескрипторов (GDT);
- *многосегментная*. На каждую программу выделяется собственная таблица локальных дескрипторов (LDT).

Несмотря на кажущуюся сложность, программисту обычно не требуется беспокоиться о сегментации.

Важное нововведение – *страничная организация памяти*. Так, сегмент памяти делится порциями по 4096 байт, называемыми *страницами*. Это позволяет избежать нехватки оперативной памяти с помощью задействования физической (локального диска). Такой подход получил название *виртуальной памяти*.

Сравним основные отличия IA-16 и IA-32:

IA-16	**IA-32**
Регистры 16-битные	Регистры 32-битные
Реальный режим адресации	Защищённый режим, виртуальный режим с возможностью эмуляции реального
Однозадачность	Многозадачность (многопоточность)
Программирование на уровне прерываний	Программирование с использованием Windows API
Сегментация памяти, ограничение ОЗУ до 1Мб	Линейная модель, доступно до 4Гб ОЗУ на одну программу

Регистры

В первой части практикума было отмечено, что новые регистры получают префикс «Е». При этом регистры общего назначения, индексные регистры и регистры-указатели (кроме IP: он полностью заменен на EIP) сохраняют в своей структуре 16-битных предшественников.

К сегментным регистрам добавлены FS и GS. Все шесть регистров теперь выполняют роль селекторов (в работе мы их не используем).

Регистр флагов EFLAGS пополнен флагами для работы с многозадачностью.

Быстрый переход на инструкции IA-32

Несмотря на серьезное отличие в схемах работы 16- и 32-битных программ, переход на 32-битную архитектуру ассемблерного кода весьма прост. Если читатель ознакомился с основными инструкциями ассемблера (ч. 2), то для быстрой «адаптации» достаточно дополнить знания расширением уже изученных инструкций.

Описание переменных и констант

Все изученные директивы описания переменных и констант полностью сохраняются. Иными словами, все 12 директив (9 для целочисленных и 3 вещественных) описания типа используются и в IA-32.

Команды пересылки данных и арифметические операции

Команды пересылки данных и арифметические команды полностью сохраняют функционал и расширяют его на случай 32-битных операндов.

Команды умножения и деления за счет 32-битных регистров дополняются новыми возможностями

```
mul / imul
```

При умножении двойных слов (один из множителей фиксируется в EAX) результат записывается в пару EDX:EAX.

Пример:

```
mov   eax,20000000h
mov   ebx,300h
mul   ebx                 ; EDX:EAX:= 60:00000000h
```

```
div / idiv
```

Если делимое находится в паре EDX:EAX, то в EDX запишется остаток, в EAX – неполное частное.

Пример:

```
mov   eax,3h
mov   edx,20h               ; EDX:EAX = 20:00000003h
mov   ebx,10000000h
div   ebx                   ; EDX:=3h, EAX:=200h
```

Условный и безусловный переход. Циклические операции

Команды условного и безусловного перехода полностью сохраняют свой функционал.

Команда jecxz метка переходит по метке, если ECX = 0.

Оператор цикла loop использует в качестве счетчика регистр ECX.

Косвенная адресация

В качестве адресных регистров используются уже 32-битные ESI, EDI, EBP, EBX. Кроме того, теперь можно использовать EAX, ECX и EDX. Директива OFFSET возвращает 4-байтовый адрес. Все возможности синтаксиса косвенной ссылки и индексации массивов сохраняются.

Пример:

```
.data
MsgEnd      BYTE    'End. Press Enter...'
.code
      mov   ebx,OFFSET MsgEnd
      mov   al,[ebx]          ; AL:='E'
      mov   al,[ebx+1]        ; AL:='n'
      mov   al,[ebx+2]        ; AL:='d'
      mov   al,MsgEnd         ; AL:='E'
      mov   al,MsgEnd+1       ; AL:='n'
      mov   al,MsgEnd+2       ; AL:='d'
      mov   al,MsgEnd[0]      ; AL:='E'
      mov   al,MsgEnd[1]      ; AL:='n'
      mov   al,MsgEnd[2]      ; AL:='d'
```

Операторы указания и расширения типа

Оператор указания типа PTR сохраняет функционал.

Пример:

```
.data
 n      DWORD    12345678h
.code
        mov    al,BYTE PTR [n]           ; AL:=78h
        mov    bx,WORD PTR [n]           ; BX:=5678h
        mov    cx,WORD PTR [n+2]         ; CX:=1234h
        mov    BYTE PTR [n+3],99h        ; n:=99345678h
```

В защищенный режим добавляется оператор расширения типа cdq, расширяющий содержимое EAX до пары EDX:EAX.

Пример:

```
mov    eax,11223344h
cdq                  ; EDX:EAX:= 00000000:11223344h
```

Операции со стеком

Стек в защищенном режиме имеет ширину 4 байта. Команды push и pop способны работать со значениями в 2 или 4 байта. Команды pusha и popa позволяют сохранять и восстанавливать значения регистров EAX, ECX, EDX, EBX, EBP, ESI, EDI. Команды pushfd и popfd сохраняют и восстанавливают значение регистра флагов EFLAGS.

Прерывания

В защищенном режиме вместо таблицы векторов прерываний используется таблица дескрипторов прерываний (IDT), реализуемых в форме шлюзов. Каждый вектор прерывания связывается с дескриптором процедуры или задачи. Вместо вызова прерывания удобнее использовать API операционной системы.

Рассмотренные вопросы на самом деле достаточно полно характеризует особенности ассемблерного кода 32-битных архитектур. Дальнейшие занятия посвящены изучению основ технологии Windows API в контексте MASM.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Чем отличается реальный режим работы процессора от защищенного? Какие новые возможности были внедрены с появлением 32-битных процессоров?

2. Опишите основные сходства и различия в ассемблерных инструкциях при программировании на 32-битных платформах.
3. Какие новые возможности появляются при работе с косвенной адресацией?

Практическое занятие № 3.2
ТЕХНОЛОГИЯ WINDOWS API

1. ЦЕЛИ РАБОТЫ

- Изучить понятие Windows API.
- Познакомиться с технологией работы динамически подключаемых библиотек.
- Разобрать шаблон консольного 32-битного приложения с минимальным DLL-набором.
- Познакомиться с отладчиком OllyDbg.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Интерфейс программирования приложений (API) *– набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.*

API могут быть предназначены для операционных систем, графических и звуковых интерфейсов, веб-ресурсов и т. д.

Основная идея сводится к следующему. Библиотеку API рассматривают как некоторый «черный ящик» с набором функций, классов и др. Компоненты API связаны между собой определенными отношениями иеррархии и взаимосвязи. Реализация компонент (как правило) скрыта от пользователя; он взаимодействует только с компонентами. Все, что необходимо программисту для успешного использования:

- понимать синтаксис API;
- изучить особенности взаимодействия компонент API;
- уметь использовать справочные материалы.

Остановимся на API операционной системы Windows. Сразу необходимо отметить, что изначально Windows API спроектирован

для использования в языке Си. Однако многие языки программирования поддерживают работу с API, например C++, Delphi, Visual Basic и т. д.

MASM не ориентировался на разработку приложений с использованием Windows API, однако такая возможность была добавлена. Серьезное развитие проекту оказало создание пакета MASM32, содержащего библиотеку функций с упрощенным синтаксисом, большой набор макросов и весьма подробное описание работы с многочисленными примерами.

DLL

Операционные системы Windows используют в своей основе технологию *динамически подключаемых библиотек*.

DLL *(англ.* ***Dynamic Link Library*** *– «библиотека динамической компоновки», «динамически подключаемая библиотека») – динамическая библиотека, позволяющая многократное использование различными программными приложениями*.

Изначально предполагалось, что технология должна обеспечить эффективное использование памяти различными приложениями.

Допустим, в определённый момент обрабатывается несколько приложений (процессов). Очевидно, что различные приложения часто обращаются к одинаковым функциям. Каждому приложению в таком случае необходимо выделить в оперативной памяти участок, хранящий информацию о вызываемых функциях. Так память очень быстро заполняется копиями данных.

С другой стороны, рациональнее вызывать функции по мере необходимости. Приложению лишь достаточно знать ссылку на файл, хранящий описание требуемых функций. Говорят, что память выделяется *динамически*, т. е. во время выполнения программы (в отличие от статического, когда информация о функциях включается в исполняемый файл во время компиляции программы).

Таким образом, приложение может использовать хоть несколько десятков динамических библиотек (файлы с расширением .dll), однако на размер самого приложения это никак не отразится.

Некоторые динамические библиотеки

Windows API использует множество библиотек, однако выделим наиболее важные и известные.

Kernel32.dll	Библиотека ядра ОС. Содержит основные функции для работы с любым консольным или оконным приложением
User32.dll	Отвечает за отрисовку окон, меню и т. п.
GDI32.dll	GDI отвечает за отрисовку линий и кривых, отображение шрифтов и обработку палитры
Shell32.dll	Работа с диалоговыми окнами
msvcrt.dll	Библиотека для работы с основными функциями и методами C\C++ Run-Time
openGL32.dll	Функции обработки двумерной и трехмерной графики

Шаблон 32-битного консольного приложения

Следующая программа содержит основные разделы приложения:

```
.386
.model flat,stdcall
.stack 4096h
option casemap:none

include c:\masm32\include\kernel32.inc

includelib c:\masm32\lib\kernel32.lib

.const

.data

.data?

.code
Start:
    ;; Код программы ;;

    invoke ExitProcess, 0
end    Start
```

Несмотря на то что это ассемблерная программа, команд ассемблера в явной форме здесь нет!

Действительно, все команды программы являются директивами или метками, многие из которых унаследованы из 16-битной архитектуры.

Директива .386 задает область допустимых инструкций. Можно указать, например, .486, .586 и т. д., однако это излишне – инструкций i386 нам вполне достаточно.

Модель памяти flat устанавливает работу процессора в защищенном режиме и соответствующую ему схему адресации.

Директива OPTION с атрибутом casemap устанавливает чувствительность компилятора к регистру букв. Это необязательная опция, однако помогает программисту контролировать имена идентификаторов и функций API.

Директива INCLUDE подключает файл с прототипами функций. Обычно эти файлы имеют то же имя, что и динамическая библиотека (kernel32.dll). Директива INLUDELIB подключает специальным образом скомпонованные библиотеки (с расширением .lib), содержащие описание функций. Они требуются только на этапе компиляции; во время выполнения программы вызываются уже динамические библиотеки. Вновь разработчики позаботились о том, чтобы заголовочные файлы (.inc) описывали прототипы скомпонованных функций в библиотеках (.lib).

Тело программы реализует всего лишь одна API функция – ExitProcess. Ее задача – корректное завершение приложения[6].

Указанный код избыточен, в нашем случае его можно сократить.

Шаблон 32-битного консольного приложения

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

include    kernel32.inc
includelib kernel32.lib
```

[6] В операционной системе Windows это важная задача, поскольку процесс в общем случае может находиться в параллельном выполнении или отдельном потоке.

```
.code
Start:
        ;;  Код программы  ;;
        invoke  ExitProcess, 0
end     Start
```

В дальнейшем полные пути к файлам с прототипами и библиотеками прописываться не будут. Исключение – случаи, когда обращение происходит в другие директории.

Для компиляции программы зададим следующие команды:

```
c\masm32\bin\ml    /c /coff  program.asm
c\masm32\bin\link  /SUBSYSTEM:CONSOLE  program.obj
```

Выбор подхода

Вышеприведенный шаблон не содержит инструкций ассемблера в явной форме. На самом деле роль директив в нем весьма высока. Можно «открыть» пару команд, если отказаться от использования директивы упрощенного вызова процедуры INVOKE.

Шаблон 32-битного консольного приложения

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

include    kernel32.inc
includelib kernel32.lib

.code
Start:
        ;;  Код программы  ;;
        push   0
        call   ExitProcess
end     Start
```

Подключение файла с прототипами также упрощает задачу. В противном случае потребуется описывать прототипы всех вызываемых функций.

Шаблон 32-битного консольного приложения

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

includelib kernel32.lib

ExitProcess  PROTO  :DWORD   ; прототип функции ExitProcess

.code
Start:
      ;;  Код программы  ;;
      push   0
      call   ExitProcess
end    Start
```

Среди программистов существует два противоположных мнения. Одни считают, что директивы и макросы необходимо активно использовать, поскольку это удобно. Другие, напротив, рекомендуют использовать как можно меньше псевдокоманд. Оба мнения имеют свои достоинства и недостатки:

- код с директивами и макросами меньше по объему и хорошо читается;
- код с минимальным числом псевдокоманд проще отлаживать, поскольку он почти совпадает с дизассемблерным.

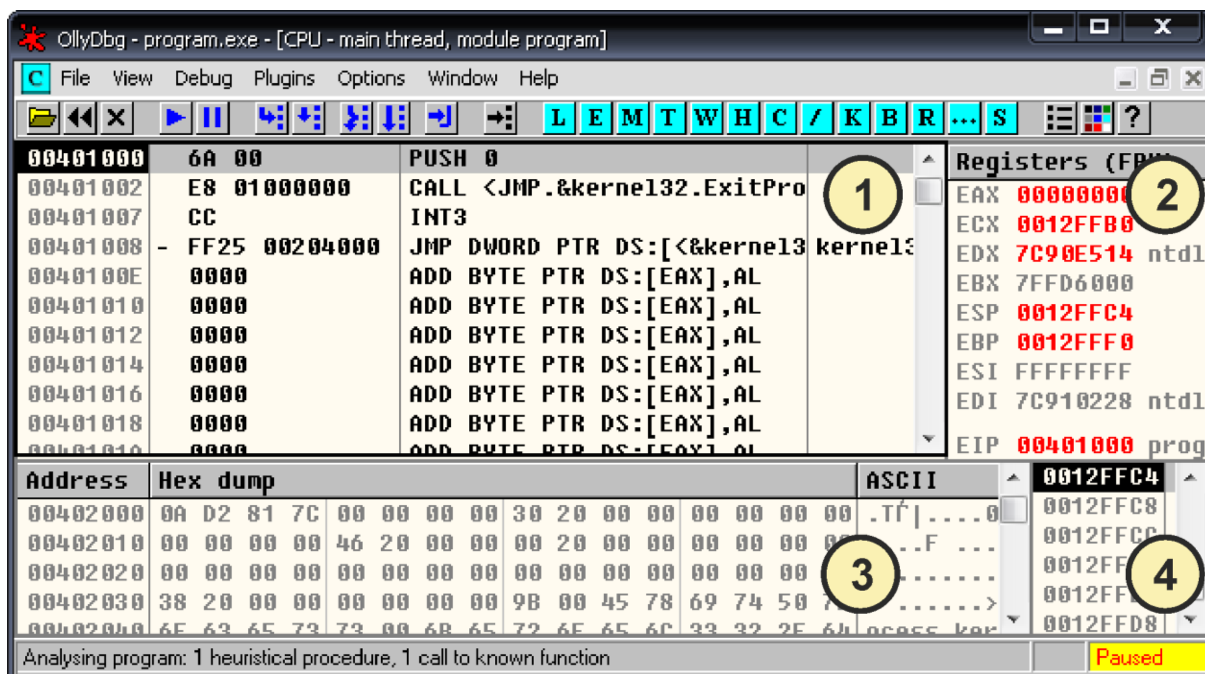
Программист может выбрать любой путь. Однако в дальнейшем мы все же будем придерживаться первого способа.

OllyDbg

OllyDbg – бесплатный 32-битный отладчик уровня ассемблера для операционных систем Windows. Отладчик способен анализировать и модифицировать уже скомпилированные программы и библиотеки. Также он отличается удобным и интуитивно понятным интерфейсом.

Основное окно отладчика:

1. Дизассемблерный код.
2. Содержимое регистров.
3. Дамп памяти.
4. Содержимое стека.



Во время отладки также присутствует консоль приложения.

Полезной функцией является анализатор кода. Во включенном режиме он группирует команды согласно инструкциям и выдает дополнительную информацию в более удобной форме (CTRL + A).

Наиболее часто вызываемые команды:

- [F7] – выполнение одного шага со входом (в циклы и процедуры);
- [F8] – выполнение одного шага с обходом (циклов и процедур);
- [F9] – выполнение всего приложения;
- [CTRL + F11] – перейти на ближайшую точку останова;
- [F2] – поставить / убрать точку останова.

В качестве примера возьмем шаблон 32-битного приложения. Перетащите исполняемый файл приложения в окно отладчика для дизассемблирования. Первое, что важно отметить – адресное пространство задается 4-байтными адресами (согласно защищенному режиму). Команда вызова процедуры ExitProcess отображается следующим образом:

```
CALL <JMP.&kernel32.ExitProcess>
```

Такая запись указывает на то, что процедура импортируется из библиотеки kernel32.dll.

Чтобы просмотреть список всех вызываемых в приложении функций, вызовем Search for / Names (CTRL + N):

Address	Section	Type	Name
00401000	.text	Export	<ModuleEntryPoint>
00402000	.rdata	Import	kernel32.ExitProcess

Двойной клик по имени функции переводит отладчик в кодовую секцию с ее описанием. Вызов окна внутри модуля отобразит функции самой библиотеки.

Кратко опишем основные элементы панели отладчика:

L E M W T C R … K

L – окно лога;
E – список модулей, которые вызывает приложение;
M – секция памяти, занятая программой;
W – отображает список окон программы;
T – список потоков программы;
C – основное окно с кодом;
R – окно ссылок, полученных при поиске в OllyDbg;
. . . – окно Run Trace;
K – информация о стеке относительно потока.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что такое API и какова его роль в программировании?
2. Что представляет собой библиотека динамической компоновки? Опишите схему работы приложения с библиотекой.
3. Перечислите основные инструкции и директивы 32-битного приложения.
4. Какими достоинствами и недостатками обладает код с директивами?
5. При первой загрузке приложения, использующего DLL библиотеки, может наблюдаться задержка. С чем это связано?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Наберите и скомпилируйте код консольного приложения, предложенного в текущем занятии.
2. Воспользуйтесь отладчиком кода OllyDbg. Добавьте простейшую функцию и проследите, как производится ее вызов?

Практическое занятие № 3.3

НЕПОСРЕДСТВЕННАЯ РАБОТА С ФУНКЦИЯМИ WINDOWS API

1. ЦЕЛИ РАБОТЫ

- Определить связь между типами Windows API и ассемблерными типами.
- Изучить особенности описания функций Windows API в MASM.
- Изучить методику работы с API функциями на примере консольного ввода/ вывода.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

От сложного к простому

Как правило, процесс изучения нового материала выстраивается от простых вещей к сложным. В нашем случае такой порядок изучения нерационален. С одной стороны, упрощенный синтаксис основан на макросах, которые сложным образом оперируют с функциями API. С другой – наша задача убедиться, что синтаксис ассемблера может быть расширен прежде всего благодаря макросам.

Именно поэтому изучение основ работы с API в ассемблере рассмотрим в следующем порядке:

1. Вызов функций Windows API в явной форме.
2. Использование библиотеки MASM32.
3. Использование макросов библиотеки MASM32.

Стандарты описания функций Windows API

При описании библиотечных функций, классов и констант требуется соблюдать определенные стандарты, чтобы программисты могли свободно ориентироваться по справочному материалу. Язык программирования также строго стандартизирован.

Программисту, даже знакомому с языком Си (C++), достаточно сложно сориентироваться в API на первоначальном этапе. Однако язык ассемблера даже упрощает работу со многими компонентами и делает ее более прозрачной, чем аналогичные манипуляции в языке Си.

Выделим наиболее важные стандарты, переориентированные на ассемблер. Согласно стандарту языка С (С++) функция/процедура Windows API:

1. Обладает уникальным именем.
2. Имеет тип (определяется типом возвращаемого значения):
   - void – функция не возвращает значение, т. е. это процедура;
   - любой кроме void – значение записывается в регистр EAX.
3. Если имеет параметры, то для каждого из них указывается:
   - роль ([IN] – для чтения; [OUT] – для записи; [RESERVED] – зарезервирован);
   - тип;
   - имя аргумента.

Роль в ассемблерном коде не указывается: префикс нам лишь поясняет, предназначен ли параметр только для чтения (создается его копия), для возврата значения из функции либо параметр зарезервирован операционной системой для своих целей.

Описание функций можно найти в справочниках, однако наиболее полная форма предоставляется компанией Microsoft в форме ресурса поддержки разработчиков:

http://msdn.microsoft.com/ru-RU/

Проанализируем компоненты на примере функции WriteConsole:

```
 BOOL WriteConsole(
    [IN]            HANDLE        hConsoleOutput,
    [IN]            CONST VOID    *lpBuffer,
    [IN]            DWORD         nNumberOfCharsToWrite,
    [OUT]           LPDWORD       lpNumberOfCharsWritten,
    [RESERVED]      LPVOID        lpReserved
)
```

Из описания (взято из справочника) можно отметить следующее:

- имя функции – WriteConsole;
- число параметров – 5 (3 для чтения, 1 для записи, 1 зарезервирован ОС).

Первый столбец указывает роль параметра, второй – тип, третий – имя. Разумеется, параметры записаны в разных строчках исключительно ради удобства, тем более что современные компиляторы без проблем распознают такую запись.

Типы Windows API

После краткого разбора функции WriteConsole возникает естественный вопрос: как ассемблер способен работать с такими типами, как, например, BOOL, HANDLE, LPDWORD и т. д.?

Чтобы подогреть интерес к вопросу, укажем, что Windows API определяет сотни различных типов!

Ответ следующий: многие типы могут быть сведены всего лишь к нескольким ассемблерным: BYTE, WORD и, прежде всего, DWORD. Устанавливать соответствие между типами позволяет директива typedef:

```
новый_тип   typedef   старый_тип
```

Указанная директива создает новый идентификатор, который полностью копирует функционал старого типа, но имеет другое имя. Все соответствия можно просмотреть в файле windows.inc:

C:\masm32\include\windows.inc

Например, в файле есть строка

```
CHAR      typedef   BYTE
```

Это означает, что вместо имени BYTE можно использовать имя CHAR. Иными словами, следующие строки равнозначны для ассемблера:

```
Mess   BYTE   "Hello, WinAPI!"
Mess   CHAR   "Hello, WinAPI!"
```

Назначение новых типов необходимо прежде всего программисту. Имя типа раскрывает его суть и, следовательно, предназначение.

Так, CHAR – «символ». Таким образом переменная Mess – совокупность символов, т. е. текстовая строка. Это звучит более естественно, чем называть текстовую строку массивом байт.

Приведем наиболее распространенные типы данных и соответствующие им ассемблерные:

C++	MASM
CHAR / UCHAR	BYTE
BOOLEAN	BYTE
USHORT	WORD
BOOL	DWORD
INT / UINT / LONG / ULONG	DWORD
HANDLE	DWORD
INT64 / UINT64 / LONG64 / ULONG64	QWORD
FLOAT	REAL4
DOUBLE	REAL8

Типы-указатели выделяются отдельно. В защищённом режиме IA-32 указатель – всегда 4-байтовая переменная. Обратите внимание, что имена типов начинаются с префикса «LP» («Long Pointer» – длинный указатель):

C++	MASM
LPBYTE	DWORD
LPBOOL	DWORD
LPWORD / LPINT / LPLONG	DWORD
LPVOID	DWORD
LPHANDLE	DWORD

Функции Windows API

Библиотека Windows API содержит сотни различных функций. Наша задача – научиться использовать функции в контексте ассемблера на примере наиболее часто вызываемых. Очевидно, что в консольных приложениях нередко требуется вывод и ввод информации с клавиатуры.

Рассмотрим следующие функции и примеры их использования:

- ExitProcess;
- GetStdHandle;
- WriteConsole;
- ReadConsole.

Отметим, что перечисленные функции входят в библиотеку kernel32.dll.

Процедура ExitProcess

```
VOID  ExitProcess (
   [IN]   UINT   uExitCode
)
```

Функция завершает работу процесса и его потоков.

Параметры

uExitCode	Код выхода для процесса или потока. Обычно равен 0 (или NULL).

Возвращаемое значение

Функция не возвращает значение.

Задача 1

Написать консольное приложение, использующее функцию ExitProcess.

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

include    windows.inc  ; здесь описаны константы WinAPI
                        ; (в частности NULL = 0)
include    kernel32.inc ; здесь описаны прототипы функций
                        ; для kernel32.dll
includelib kernel32.lib ; подключаем библиотеку импорта

.code
Start:
    invoke  ExitProcess, NULL    ; вызываем функцию

end   Start
```

Функция GetStdHandle

Многие функции для корректной работы требуют *дескриптор* (описатель). Это связано с особенностью архитектуры операционной системы Windows. Поскольку ей приходится анализировать работу одновременно с несколькими приложениями, то необходимо уметь отличать различные процессы: вывод данных, ввод, работа с файлами, прорисовка и т. д. Для этого подобным функциональным элементам Windows назначает дескриптор. По существу это уникальное число. Задача программиста – указать, какой дескриптор требуется получить.

Как правило, полученный дескриптор записывают в переменную и используют в дальнейшем.

```
HANDLE GetStdHandle (
  [IN]   DWORD  nStdHandle
)
```

Функция GetStdHandle извлекает дескриптор для стандартного ввода данных, стандартного вывода или стандартной ошибки устройства.

Параметры

nStdHandle Стандартное устройство, для которого дескриптор должен быть возвращен. Этот параметр может иметь значение:

- STD_INPUT_HANDLE – дескриптор стандартного устройства ввода;
- STD_OUTPUT_HANDLE – дескриптор стандартного устройства вывода;
- STD_ERROR_HANDLE – дескриптор стандартной ошибки устройства.

Возвращаемое значение

При успешном выполнении в регистр EAX записывается требуемый дескриптор. Многие дескрипторные типы часто достаточно просто отличить от обычных: они содержат ключевое «handle» в своем названии.

Задача 2

Написать консольное приложение, получающее дескрипторы ввода и вывода и записывающее их в отдельные переменные.

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

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

.data
InHandle    HANDLE  ?       ; дескриптор для ввода
OutHandle   HANDLE  ?      ; дескриптор для вывода

.code
Start:
        ; Получаем в EAX дескриптор ввода и записываем его
        ; в переменную InHandle.
        invoke  GetStdHandle, STD_INPUT_HANDLE
        mov     InHandle,eax

        ; Получаем в EAX дескриптор вывода и записываем его
        ; в переменную OutHandle.
        invoke  GetStdHandle, STD_OUTPUT_HANDLE
        mov     OutHandle,eax

        invoke  ExitProcess, NULL
end    Start
```

Функция WriteConsole

```
BOOL WriteConsole (
  [IN]          HANDLE      hConsoleOutput,
  [IN]          LPCHAR      *lpBuffer
  [IN]          DWORD       nNumberOfCharsToWrite
  [OUT]         LPDWORD     lpNumberOfCharsIsWritten
  [RESERVED]    LPVOID      lpReserved
)
```

Функция WriteConsole выводит строку из буфера на экран.

Параметры

hConsoleOutput	– дескриптор экранного буфера;
*lpBuffer	– указатель на буфер (переменную), хранящий символы;
nNumberOfCharsToWrite	– число символов для вывода;
lpNumbersOfCharsIsWritten	– указатель на число отображенных символов;
lpReserved	– должен быть равен нулю (NULL).

Возвращаемое значение

При успешном выполнении в регистр EAX записывается ненулевое значение, иначе нуль.

Задача 3

Написать консольное приложение, выводящее строку на экран.

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

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

.data
Mess       CHAR    "Hello, Assembler!"       ; строка
OutHandle  HANDLE   ?                        ; дескриптор вывода
cWritten   DWORD    ?       ; число выведенных по факту символов

.code
Start:
        ; Получаем в EAX дескриптор вывода и записываем его
        ; в переменную OutHandle.
        invoke  GetStdHandle, STD_OUTPUT_HANDLE
        mov     OutHandle,eax

      invoke  WriteConsole,
              OutHandle,        ; дескриптор вывода
              ADDR Mess,        ; адрес строки
```

```
        SIZEOF Mess,      ; вывод всех символов
        ADDR cWritten,    ; указатель на число
                          ; выведенных символов
        NULL              ; нуль

    invoke  ExitProcess, NULL
end    Start
```

Поскольку задержки консоли пока не организовано, то результат работы полезно проследить в отладчике. Прежде всего включим режим анализатора (CTRL + A), который сгруппируют функции и передаваемые параметры

```
00401000 ┌$  6A F5         PUSH -0B                 ┌DevType = STD_OUTPUT_HANDL
00401002 │.  E8 2B000000   CALL <JMP.&kernel32.G    └GetStdHandle
00401007 │.  A3 11304000   MOV DWORD PTR DS:[403
0040100C │.  6A 00         PUSH 0                   ┌pReserved = NULL
0040100E │.  68 15304000   PUSH program2.0040301    │pWritten = program2.00403
00401013 │.  6A 11         PUSH 11                  │CharsToWrite = 11 (17.)
00401015 │.  68 00304000   PUSH program2.0040300    │Buffer = program2.0040300
0040101A │.  FF35 1130400  PUSH DWORD PTR DS:[40    │hConsole = NULL
00401020 │.  E8 13000000   CALL <JMP.&kernel32.W    └WriteConsoleA
00401025 │.  6A 00         PUSH 0                   ┌ExitCode = 0
00401027 └.  E8 00000000   CALL <JMP.&kernel32.E    └ExitProcess
0040102C  .- FF25 0820400  JMP DWORD PTR DS:[<&k    kernel32.ExitProcess
```

Функция ReadConsole

```
BOOL ReadConsole (
  [IN]       HANDLE   hConsoleInput,
  [OUT]      LPVOID   lpBuffer
  [IN]       DWORD    nNumberOfCharsToRead
  [OUT]      LPDWORD  lpNumberOfCharsRead
  [RESERVED] LPVOID   lpReserved
)
```

Функция ReadConsole вводит символы в буфер.

Параметры

hConsoleInput	– дескриптор консольного буфера ввода;
lpBuffer	– указатель на буфер, принимающий введенные символы;
nNumberOfCharsToRead	– число вводимых символов;
lpNumbersOfCharsRead	– указатель на число фактически считанных символов;
lpReserved	– должен быть равен нулю (NULL).

Возвращаемое значение
При успешном выполнении в регистр EAX записывается ненулевое значение, иначе нуль.

Задача 4

Написать консольное приложение, считывающее вводимую строку в переменную.

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

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

.data?
Buffer     CHAR     100  dup(?) ; буфер на 100 символов
InHandle   HANDLE   ?           ; дескриптор ввода
cRead      DWORD    ?           ; число введенных по
                                ; факту символов

.code
Start:
      ; Получаем в EAX дескриптор ввода и записываем его
      ; в переменную InHandle.
      invoke  GetStdHandle, STD_INPUT_HANDLE
      mov     InHandle,eax

      invoke  ReadConsole,
              InHandle,          ; дескриптор ввода
              ADDR Buffer,       ; адрес буфера
              SIZEOF Buffer,     ; размер буфера
              ADDR cRead,        ; указатель на число
                                 ; введенных символов
              NULL               ; нуль

      invoke  ExitProcess, NULL
end    Start
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Каким образом удается согласовать множество различных API-типов с ассемблерными?
2. Опишите основные элементы API функции.
3. Что такое дескрипторы и для чего они применяются?
4. Почему параметры функции, возвращающие из нее значение, являются указателями?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Опишите переменную знакового типа. Программно определить и вывести на экран, является ли это число положительным, отрицательным либо нулем. Организуйте с помощью функции ReadConsole задержку консоли перед закрытием.
2. Найдите описание функции wsprintf. С ее помощью осуществить вывод содержимого регистров EAX и EBX.

Практическое занятие № 3.4

БИБЛИОТЕКА MASM32

1. ЦЕЛЬ РАБОТЫ

- Изучить некоторые дополнительные возможности библиотеки MASM32.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Упрощенные функции

В настоящий момент Macro Assembler больше не является коммерческим продуктом, однако поддерживается компанией Microsoft и входит в состав Visual C++. Серьезное развитие ассемблер получил за счет поддержки большого сообщества программистов.

Одним из основателей пакета MASM32 как надстройки над классическими версиями MASM 6.14 / 6.15 считается Стив Хатчиссон. В частности, он разработал библиотеку импорта MASM32.lib, включающую наиболее часто вызываемые функции, описанные в упрощенной форме (по сравнению с аналогами Windows API). Как отмечает автор, библиотека ориентировалась на упрощение программирования приложений с использованием ассемблера.

Файл с прототипами функций и сама библиотека находятся в отдельной директории:

c:\masm32\m32lib\masm32.inc

c:\masm32\m32lib\masm32.lib

Необходимо учитывать, что одна функция библиотеки нередко запрашивает несколько функций из стандартных API библиотек.

Рассмотрим следующие процедуры библиотеки:

- ClearScreen;
- StdOut;
- StdIn;
- locate.

Процедура ClearScreen

ClearScreen PROC

Процедура очищает экран консоли.

Параметры

Процедура не имеет параметров.

Возвращаемое значение

Процедура не возвращает значение.

Процедура StdOut

StdOut PROC lpszText:DWORD

Процедура выводит строку на экран. Строка должна оканчиваться нулевым байтом.

Параметры

lpszText – указатель на строку, оканчивающуюся нулем.

Возвращаемое значение

Процедура не возвращает значение.

Процедура StdIn

StdIn PROC lpszBuffer:DWORD, bLen:DWORD

Процедура записывает введенный текст в буфер до нажатия Enter.

Параметры

lpszBuffer – указатель на буфер, в который будет записан текст;

bLen – длина буфера.

Возвращаемое значение

Процедура не возвращает значение.

Процедура locate

locate PROC x:DWORD, y:DWORD

Процедура перемещает курсор в место с указанными координатами.

Параметры

x – номер столбца;

y – номер строки.

Возвращаемое значение

Процедура не возвращает значение.

Задача 1

Написать консольное приложение, запрашивающее ввод текста. Далее этот текст выводится на экран. Обеспечить задержку консоли перед выходом.

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

include windows.inc
include c:\masm32\m32lib\masm32.inc   ; прототипы и константы
                                      ; библиотеки
include kernel32.inc

includelib c:\masm32\m32lib\masm32.lib  ; библиотека
includelib kernel32.lib

BUFSIZE   EQU     100                 ; максимальный размер буфера

.data
InputTXT  CHAR    "Text: ", 0h        ; строка, оканчивающаяся
                                      ; нулем
.data?
Buffer    CHAR    BUFSIZE  DUP(?)     ; буфер для ввода
```

```
.code
Main   proc
       invoke  ClearScreen
       invoke  StdOut,
               ADDR InputTXT        ; пояснение перед вводом
       invoke  StdIn,
               ADDR Buffer,
               BUFSIZE              ; вводим текст в буфер
       invoke  StdOut,
               ADDR Buffer          ; выводим введенный текст
       invoke  StdIn,
               ADDR Buffer,
               1                    ; ждем нажатия клавиши
       invoke  ExitProcess, NULL
Main   endp
end    Main
```

В режиме отладки можно убедиться, ввод StdIn и вывод StdOut запрашивают ряд других функций, в частности это ReadFile и WriteFile:

00402000	.rdata	Import	kernel32.ExitProcess
00402014	.rdata	Import	kernel32.FillConsoleOutputCharacterA
00402018	.rdata	Import	kernel32.GetConsoleScreenBufferInfo
00402004	.rdata	Import	kernel32.GetStdHandle
00401000	.text	Export	<ModuleEntryPoint>
0040200C	.rdata	Import	kernel32.ReadFile
0040201C	.rdata	Import	kernel32.SetConsoleCursorPosition
00402010	.rdata	Import	kernel32.SetConsoleMode
00402008	.rdata	Import	kernel32.WriteFile

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что представляет собой библиотека MASM32? Для каких целей она разработана?
2. Найдите в папке с ассемблером файл справки masmlib.chm и исследуйте его содержимое. Какие полезные возможности включены в пакет?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Определить, существует ли треугольник с заданными сторонами. Для вывода текста используйте функцию StdOut.
2. Найдите описание функций конвертирования числа в строку и обратно (dwtoa/atodw). Задайте массив чисел и реализуйте поиск и вывод количества положительных элементов в нем.

Практическое занятие № 3.5

БИБЛИОТЕКА МАКРОСОВ MASM32

1. ЦЕЛЬ РАБОТЫ

- Изучить некоторые возможности расширенной библиотеки макросов MASM32.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Макрос как высокоуровневая инструкция

Помимо библиотеки MASM32.lib в последние версии ассемблера включен достаточно внушительный пакет макросов, упрощающий синтаксис вызова стандартных функций. В определенных случаях программный код, использующий эти макросы, сравним по компактности и простоте с программой, написанной на языке C/C++.

Для использования макросов достаточно подключить соответствующий файл:

```
include    c:\masm32\macros\macros.asm
```

Макросы могут обращаться к нескольким функциям, расположенным в разных библиотеках, поэтому не забудьте подключить требуемые библиотеки. Кроме того, не забывайте, что макросы не заботятся о сохранении значений регистров: при необходимости делайте это самостоятельно.

Рассмотрим следующие макросы:

- cls;
- print;
- input;
- inkey;
- exit;

- printf;
- ustr$ / sstr$;
- uval / sval.

Макрос cls

cls

Макрос очищает консоль.

Параметры

Не имеет.

Возвращаемое значение

Не возвращает.

Макрос print

```
print  lpText
print  lpText, 13, 10
print  "The text"
print  "The text", 13, 10
```

Макрос выводит строку, текст в кавычках и/или последовательность байт на экран.

Параметры

Указатель на строковую переменную, текст в кавычках и/или байты.

Возвращаемое значение

Не возвращает.

Макрос input

```
mov    lpText, input()
mov    lpText, input("Text: ")
```

Макрос получает введенную строку и записывает ее адрес в переменную lpText.

Параметры

Подсказка для ввода в кавычках либо без нее.

Возвращаемое значение

В регистр EAX запишется число введенных символов.

Макрос inkey

```
inkey
inkey  NULL
inkey  "Press a key to continue..."
```

Макрос осуществляет задержку процесса до нажатия любой клавиши.

Параметры

Нуль, текст-подсказка либо может не иметь.

Возвращаемое значение

В регистр EAX запишется сканкод нажатой клавиши.

Макрос exit

```
exit
```

Макрос завершает приложение или процесс.

Параметры

Код ошибки (в данной книге не рассматривается).

Возвращаемое значение

Не возвращает.

Задача 1

Приведите примеры использования рассмотренных выше макросов.

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

include windows.inc
include c:\masm32\macros\macros.asm   ; подключаем макросы
include c:\masm32\m32lib\masm32.inc
include kernel32.inc
include msvcrt.inc

includelib c:\masm32\m32lib\masm32.lib
includelib kernel32.lib
includelib msvcrt.lib

NEWLINE   EQU   13,10
```

```
.data
Text   CHAR     "This is Assembler.", 0h
lpBuf  DWORD    ?

.code
Start:
      cls                                   ; очищаем консоль
      print  "Hello, Denis!",
             NEWLINE                        ; выводим сообщение
      print  ADDR Text,
             NEWLINE                        ; выводим сообщение
      mov    lpBuf,
             input("Input a text: ")        ; ввод с подсказкой
      print  "You have written: "           ; выводим сообщение
      print  lpBuf,
             NEWLINE                        ; выводим введенный текст
      inkey  "Press any key to exit..."     ; ожидаем нажатия
                                            ; клавиши
      exit                                  ; завершаем процесс
end   Start
```

Макрос printf

printf ("Форматирующая строка",значения)

Макрос осуществляет вывод значений согласно специальным символам в форматирующей строке.

Параметры

Форматирующая строка содержит специальные символы для форматного вывода строк и чисел.

Некоторые управляющие символы

\n – переход на новую строку;

\t – табуляция;

%d – подстановка целого числа;

%s – подстановка строки.

Возвращаемое значение

Не возвращает.

Макрос printf – одноименный аналог популярной среди программистов C/C++ функции форматированного вывода на консоль. В следующем примере демонстрируется универсальность макроса.

Задача 2

Продемонстрировать примеры вывода строк и чисел с помощью макроса printf.

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

include windows.inc
include c:\masm32\macros\macros.asm
include c:\masm32\m32lib\masm32.inc
include kernel32.inc
include msvcrt.inc

includelib c:\masm32\m32lib\masm32.lib
includelib kernel32.lib
includelib msvcrt.lib

.data
num1   DWORD  999
num2   DWORD  888

.code
Start:
        printf  ("PRINTF():")
        printf  ("\n\nYakubovich\nDenis\nAndreevich\n")
        printf  ("Tab1\tTab2\tTab3\n\n")
        printf  ("EAX = %d\n",eax)
        printf  ("%d\t%d\n\n",num1,num2)
        inkey   "Press a key to exit..."
        exit
end     Start
```

```
C:\masm32\projects\examples\ma...
PRINTF():

Yakubovich
Denis
Andreevich
Tab1     Tab2     Tab3

EAX  = 18
999      888

Press a key to exit...
```

Однако не стоит забывать, что printf работает с числами типа DWORD (SDWORD). Для вывода чисел меньшего типа требуется расширение до двойного слова.

Макросы ustr$ и sstr$

```
mov  lpNum,ustr$(число)

mov  lpNum,sstr$(число)
```

Макросы переводят целое число (без знака/со знаком) в строку и возвращают указатель на эту строку.

Параметры

Непосредственно заданное число или переменная.

Возвращаемое значение

Не возвращает.

Задача 3

Задать два противоположных числа, конвертировать их в строку и вывести на экран.

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

include windows.inc
include c:\masm32\macros\macros.asm
include c:\masm32\m32lib\masm32.inc
include kernel32.inc
include msvcrt.inc
```

```
includelib c:\masm32\m32lib\masm32.lib
includelib kernel32.lib
includelib msvcrt.lib

.data
lpNum1 DWORD  ?     ; указатель на первое число
lpNum2 DWORD  ?     ; указатель на второе число
lpNum  DWORD  ?     ; указатель на число в переменной
num    DWORD  12345

.code
Start:
    mov    lpNum1,ustr$(123)
    mov    lpNum2,sstr$(-123)
    mov    lpNum,ustr$(num)
    printf  ("%s and %s\n",lpNum1,lpNum2)
    printf  ("%s\n",lpNum)
    inkey  "Press any key to exit..."
    exit
end   Start
```

Макросы uval и sval

```
mov  num,uval(lpBuffer)
mov  num,sval(lpBuffer)
```

Макросы преобразуют строку в целое число (без знака/со знаком).

Параметры

lpBuffer – указатель на строку, хранящую цифры числа.

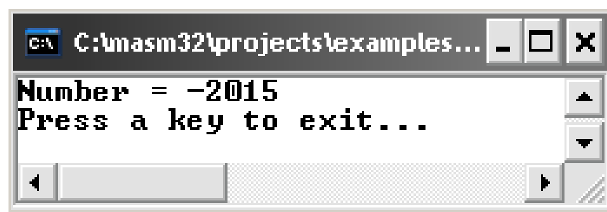
Возвращаемое значение

Не возвращает.

Задача 4.

Конвертировать заданную строку в числовую переменную.

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

include windows.inc
include c:\masm32\macros\macros.asm
include c:\masm32\m32lib\masm32.inc
include kernel32.inc
include msvcrt.inc

includelib c:\masm32\m32lib\masm32.lib
includelib kernel32.lib
includelib msvcrt.lib

.data
buff   CHAR    "-2015"  ; строка, хранящая число со знаком
num   DWORD   ?         ; переменная, зарезервированная под число

.code
Start:
      mov     num,sval(ADDR buff)
      printf  ("Number = %d\n",num)
      inkey   "Press a key to exit..."
      exit
end    Start
```



Masm32RT в помощь новичку

Начинающему на ассемблере программисту сложно сориентироваться в наборе API функций и библиотек. Еще сложнее определить, каким образом связаны функции в макросах (это требует немалого времени и длительной практики написания кода).

Чтобы упростить задачу, в пакет включен файл с прототипами MASM32RT.inc. В этом файле подключены ссылки на наиболее часто вызываемые библиотеки и файл с макросами, а главное – учитывается связь между ними. Кроме того, в нем указаны начальные директивы, которые мы использовали во всех программах этого раздела. В случае необходимости файл можно дополнить ссылками на другие библиотеки. Как и следует ожидать, подключение этого файла автоматически добавляет необходимый код к файлу с программой в процессе компиляции.

Задача 5

Перепишите задачу 4, используя файл MASM32rt.inc.

```
include masm32rt.inc

.data
buff  CHAR   "-2015"  ; строка, хранящая число со знаком
num   DWORD  ?        ; переменная, зарезервированная под число

.code
Start:
    mov    num,sval(ADDR buff)
    printf ("Number = %d\n",num)
    inkey  "Press a key to exit..."
    exit
end  Start
```

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Какие возможности предоставляют макросы библиотеки MASM32?
2. Перечислите изученные макросы, какие примеры их использования вы можете привести?
3. Проведите сравнительный анализ возможностей ввода/вывода с помощью функций API, упрощенных функций библиотеки MASM32 и макросов.
4. Какие «скользкие моменты» необходимо учитывать, используя упрощенные макросы?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. С помощью макросов осуществить вывод элементов заданного массива типа DWORD. Как изменить программу, чтобы можно было выводить и элементы типа BYTE, WORD?
2. Напишите процедуру сортировки выбором массива. Прототип процедуры:

   SortMass PROTO DWORD PTR Mass, DWORD n,

   где первый параметр – указатель на массив, второй – число элементов массива. Также описать процедуру вывода массива и продемонстрировать массив до и после сортировки.

Практическое занятие № 3.6

СОЗДАНИЕ ДИНАМИЧЕСКИХ БИБЛИОТЕК

1. ЦЕЛЬ РАБОТЫ

- Изучить основы технологии библиотек динамической компоновки и методику их разработки средствами ассемблера.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Технология работы с DLL

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

Для MS-DOS эта ситуация не была проблемной, поскольку в определенный момент времени могла выполняться только одна программа. Однако в системе Windows в фиксированный промежуток времени могут выполняться десятки и даже сотни различных приложений или процессов (не говоря о многоядерных вычислениях). В этом случае оперативная память будет перегружена.

Все это привело к появлению технологии динамических библиотек DLL. Windows не загружает несколько копий библиотеки в

память, а позволяет процессам разделять один и тот же код этой DLL. Впрочем, секция данных копируется для каждого процесса.

В отличие от статических библиотек DLL вызывается только в процессе выполнения программы. Библиотеку также можно и выгрузить из памяти во время выполнения программы, если она больше не нужна[7].

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

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

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

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

DLL без точки входа

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

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

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

[7] Однако память освободится сразу в случае, если все остальные процессы также больше не обращаются к библиотеке.

Общий механизм создания и подключения динамической библиотеки следующий:

1. Создается файл с кодом библиотеки.
2. В другом специальном файле указывается, какие функции берутся из библиотеки.
3. Файлы компилируются в библиотеку.
4. Основная программа компилируется со ссылкой на библиотеку.

Создадим код библиотеки в файле MyDLL.asm.

MyDLL.asm.

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

include    kernel32.inc
includelib kernel32.lib

.code
SumArray   proc  lpArr: PTR DWORD, len: DWORD

        mov   ebx,lpArr ; EBX:= указатель на массив
        mov   eax,0     ; в EAX накапливаем сумму
        mov   ecx,len   ; ECX: = количество элементов
  @L:
        add   eax,[ebx]
        add   ebx,4
        loop  @L
                        ; результат в EAX
        ret
SumArray   endp
end
```

На первый взгляд код описывает вполне автономную программу. Однако это не так. Обратите внимание, что файл завершен командой end, а не end SumArray.

Это говорит о том, что функция SumArray уже не выполняет роль точки входа. Действительно, любая из функций библиотеки может стать точкой входа.

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

MyDLL.def

```
LIBRARY  MyDLL
EXPORTS  SumArray
```

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

Теперь можно скомпилировать оба файла и получить библиотеку:

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

- Ключ /DLL указывает, что компонуется динамическая библиотека.
- Ключ /DEF ссылается на карту импортируемых функций.
- Ключ /NOENTRY помечает, что в библиотеке не будет точки входа.

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

Напишем код основной программы:

DLLExample.asm

```
include masm32rt.inc
; подключаем файл импорта
includelib c:\masm32\projects\MyDLL.lib
; прототип импортируемой функции
SumArray  PROTO  :PTR DWORD, :DWORD

.data
Mass   DWORD   10, 23, 500, 100, 23, 4
Sum    DWORD   0

.code
Start:
     invoke  SumArray, ADDR Mass, LENGTHOF Mass
     mov     Sum,eax
     printf  ("Sum=%d",Sum)
     inkey   NULL
     exit
end    Start
```

Разумеется, можно создать заголовочный файл MyDLL.inc, в котором описан указанный прототип, и подключить его в программе директивной include, чтобы не писать прототипы внутри программы (однако ради одной функции мы этого не делаем).

Для компиляции этой программы дополнительных ключей не понадобится: все необходимое уже сделано ранее. Если компиляция пройдет успешно, то мы получим рабочее приложение.

DLL с точкой входа

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

MyDLL.asm

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

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

.code
DLLEntry   proc  hInstDLL: HINSTANCE,
           reason: DWORD,
           reserved: DWORD

           mov   eax,TRUE
           ret
DLLEntry   endp

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

end        DLLEntry
```

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

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

Параметр reserved всегда берут нулем (NULL).

Строка

```
mov   eax,TRUE
```

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

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

При компиляции необходимо удалить ключ /NOENTRY:

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

Поиск DLL-библиотеки

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

- директория, из которой загружено приложение;
- текущая директория;
- системная директория (обычно C:\Windows\System32);
- системная директория для 16-битных приложений (обычно C:\Windows\System);
- Windows-директория;
- директории, указанные в переменной окружения PATH.

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

Динамическое подключение

До сих пор мы рассматривали *статический способ подключения* DLL-библиотек. Это означает, что операционная система автоматически подключает библиотеку к программе. Автоматизация была возможна благодаря использованию библиотек импорта. Поэтому статическое подключение является наиболее простым в плане реализации.

Однако статическое подключение обладает рядом недостатков:

- если библиотека отсутствует, то приложение не сможет быть выполнено и экстренно завершится;
- может отсутствовать файл библиотеки импорта .lib.

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

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

- LoadLibrary – ищет и загружает библиотеку; в противном случае возвращает 0;
- GetProcAdress – осуществляет поиск функции в указанной библиотеке и возвращает указатель на нее; иначе 0;
- FreeLibruary – освобождает библиотеку.

Очевидно, что использование этих функций позволит:

- отследить отсутствие библиотеки и эффективно это обработать;
- отследить и обработать некорректность вызываемой функции;
- освободить ресурсы ОЗУ, если библиотека более не используется.

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

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

MyDLL.asm

```
include  masm32rt.inc

.const
libName  CHAR   "MyDLL", 0h            ; имя библиотеки
funcName CHAR   "SumArray", 0h         ; имя функции

.data
Mass    DWORD   10, 23, 500, 100, 23, 4
Sum     DWORD   0

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

.code
Start:
; -------------------------------------------------- ;
;  Получаем дескриптор. Если вызов успешен (библиотека
;  найдена), то сохраняем его в переменную hLib. Иначе –
;  выход с обработкой исключения.
; -------------------------------------------------- ;
    invoke  LoadLibrary, ADDR libName
    cmp     eax,0
    je      @error
; -------------------------------------------------- ;
;  Получаем указатель на функцию в библиотеке и
;  вызываем ее, но уже по адресу!
; -------------------------------------------------- ;
    mov     hLib,eax
    invoke  GetProcAddress, hLib, ADDR funcName
    mov     lpFunc,eax

    push    LENGTHOF Mass
    push    OFFSET Mass
```

```
    call   [lpFunc]
    mov    Sum,eax
    printf  ("Max=%d",Sum)
; ------------------------------------------------- ;
; Освобождаем ресурсы - больше библиотека не
; понадобится
; ------------------------------------------------- ;
    invoke  FreeLibrary, hLib
    jmp    @exit
@error:
    print  "Libruary not found!"

@exit:
    inkey  NULL
    exit
end   Start
```

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

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что такое статическая компоновка и чем она отличается от динамической?
2. Почему возникает необходимость использования динамически подключаемых библиотек?
3. В каких случаях для библиотеки может понадобиться точка входа?
4. Опишите процесс создания и подключения динамической библиотеки без точки входа.
5. Почему динамическое подключение библиотеки более безопасно?
6. В каких директориях осуществляется поиск библиотек системой Windows?

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

Создайте динамическую библиотеку MyMath. Реализуйте в библиотеке процедуры по следующим прототипам:
Module PROTO n: DWORD – возвращает в EAX модуль числа n;
MaxArr PROTO pMass: PTR DWORD, n: DWORD – возвращает максимальный элемент массива из n элементов.

Практическое занятие № 3.7
СТРУКТУРЫ И ОБЪЕДИНЕНИЯ

1. ЦЕЛЬ РАБОТЫ

- Изучить понятие структуры и области ее применения в ассемблере.

2. ТЕОРЕТИЧЕСКИЙ БЛОК

Структура как тип данных

Структуры расширяют понятие типов переменных, позволяют передавать большие данные процедурам в компактной форме, что экономит память и избавляет от ошибок из-за невнимательности.

Структуры активно используются в языках C/C++. Они являются родоначальниками классов – элементов, определивших объектно-ориентированную парадигму программирования. Иными словами, структуры расширяют возможности процедурного программирования, позволяют решать более широкие типы задач.

***Структура** – тип данных, связывающий несколько переменных в едином блоке памяти.*

Структуру можно представить как несколько ячеек памяти (в общем разного типа), которые объединены в одну. Каждая переменная называется *полем* структуры.

Определение структуры

Структура определяется директивой STRUCT:

```
Название  STRUCT
    ;; поля структуры
Название  ENDS
```

Поля структуры – внутренние переменные: они могут быть как простых, так и структурных типов.

Пусть требуется описать студента, у которого есть имя, возраст и номер зачетной книжки. Поскольку студентов может быть несколько, то рационально организовать структуру:

Каждый атрибут – поле структуры (всего их три). В MASM эту абстракцию можно описать следующим образом:

```
STUDENT   STRUCT
    ID   DWORD        ?                  ; номер зачетки
    Name  CHAR        20  DUP(?)         ; имя студента
    Age   BYTE        ?                  ; возраст
STUDENT   ENDS
```

Поля описываются как и обычные переменные. В частности, они могут быть неинициализированы по умолчанию.

Переменные структурного типа описываются следующим образом:

```
.data
; поля по умолчанию
Stud_1     STUDENT  <>
; задаем свои значения
Stud_2     STUDENT  <777,"Petrov",21>
; переопределяем только третье поле
Stud_3     STUDENT  <,,20>

; массив из трех структур
Mass       STUDENT  3  DUP(<0,,0>)
```

В угловых скобках указываются значения, которые передаются полям в соответствующем порядке. Значения можно опускать (т. е. брать по умолчанию). Приведенная выше инициализация во многом похожа на конструктор класса в C++/C#.

Обращение к полям структуры

Для обращения к полю структуры используется оператор точка.

Например, следующая команда пересылает регистру EAX значение из поля ID переменной Stud_2:

```
mov     eax,Stud_2.ID
```

С помощью смещения можно обратиться к отдельным байтам поля:

```
mov     bl,Stud_2.LName       ; BL:="P"
mov     bl,Stud_2.LName+1     ; BL:="e"
```

Поскольку данные о полях в ОЗУ записываются подряд, то вполне работоспособной будет, например, команда

```
mov     cl,BYTE PTR Stud_2.ID + 24
; равносильно  cl,Stud_2.Age
```

По аналогии можно получить адреса полей структуры:

```
mov     ebx,OFFSET Stud_2.ID
mov     ebx,OFFSET Stud_2.LName
mov     ebx,OFFSET Stud_2.Age
```

Косвенная адресация

Косвенная адресация структуры полезна прежде всего при передаче структуры в качестве параметра процедуры. Передавая указатель на структуру, в стек копируется лишь ее адрес, а не значения всех полей. Кроме того, в процедуре можно менять содержимое полей структуры:

```
mov     ebx,OFFSET Stud_2      ; EBX:=адрес структуры
mov     eax,(STUDENT PTR [ebx]).ID      ; EAX:=777
mov     al,(STUDENT PTR [ebx]).LName    ; AL:="P"
mov     ah,(STUDENT PTR [ebx]).Age      ; AH:=21
```

Важно отметить, что без оператора уточнения типа PTR MASM не сможет корректно обработать запрос

```
mov     ah,[ebx].Age      ;  ошибка!
```

Задача 1

Опишите структуру Student и задайте несколько ее экземпляров. Скопировать объекты в массив структур и вывести для каждого объекта содержимое поля ID.

```
include masm32rt.inc

STUDENT   STRUCT
    ID   DWORD    ?
    LName CHAR     20  DUP(?)
    Age   BYTE     ?
STUDENT   ENDS

.data
Stud_1     STUDENT  <776,"Ivanov",21>
Stud_2     STUDENT  <777,"Petrov",21>
Stud_3     STUDENT  <779,"Krupnov",20>

Mass       STUDENT  3  DUP(<0,,0>)

.code
Start:
        ; сбрасываем флаг направления
        cld
        ; EDI:=адрес приемника
        mov     edi,OFFSET Mass
        ; ECX:=размер одного элемента
        mov     ecx,TYPE STUDENT
        mov     esi,OFFSET Stud_1      ; ESI:=адрес источника
        rep     movsb                  ; копируем
        mov     ecx,TYPE STUDENT
        mov     esi,OFFSET Stud_2
        rep     movsb
        mov     ecx,TYPE STUDENT
        mov     esi,OFFSET Stud_3
        rep     movsb

        mov     ecx,3
        mov     edi,OFFSET Mass
@L:
        push    ecx
        printf  ("%d\n",(STUDENT PTR [edi]).ID)
```

```
    pop    ecx
    add    edi,TYPE STUDENT
    loop   @L

    inkey  "..."
    exit

end   Start
```

Код требует пояснения. Перед нами стоит задача заполнить элементы массива отдельными структурными переменными Stud_1, Stud_2, Stud_3. Очевидно, что в ассемблере она решается побайтовым копированием.

Одним из наиболее эффективных приемов считается использование операторов строкового примитива movsb. Сама команда копирует только один байт из источника в приемник (их адреса задаются в ESI и EDI). Однако в паре с командой rep (префикс повтора) копируется заданное в ECX число байт. Процедура повторяется для каждой переменной, при этом регистры ESI и EDI автоматически увеличиваются.

Что касается цикла вывода, то здесь для перехода к новому элементу указатель увеличивается на размер типа STUDENT. Заметим, что код неоптимизирован (см. п. 1 самостоятельной работы).

Объединения

Объединения похожи на структуры, однако в отличие от последних их поля имеют одинаковое смещение относительно начала, т. е. перекрываются.

Такая организация данных имеет определенные преимущества. С одной стороны, это вариативность представления данных, а с другой – экономия памяти под хранение объекта. Действительно, размер объединения будет определяться самым «большим» полем, а не суммарной размерностью всех полей.

Объединение определяется директивой UNION:

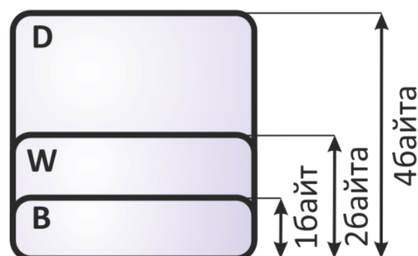
```
Название  UNION
    ;; поля объединения
Название  ENDS
```

В отличие от структуры при инициализации переменной передается только одно значение.

Например, требуется описать тип данных, предоставляющий удобный механизм работы с переменной как регистром. Объединение как нельзя лучше решает эту задачу.

Задача 2

```
include masm32rt.inc

REGISTER  UNION
   D   DWORD  0      ; 32-битная область
   W   WORD   0      ; 16-битная область
   B   BYTE   0      ; 8-битная область
REGISTER  ENDS

.data
Reg   REGISTER   <1234h>

.code
Start:
     xor     eax,eax     ; EAX := 0
     mov     al,Reg.B    ; AL := 34h
     mov     ax,Reg.W    ; AX := 1234h
     mov     eax,Reg.D   ; EAX := 00001234h
     exit
end   Start
```



Описанное объединение решает сразу несколько вопросов:

- придает требуемый функционал переменной;
- экономит память (Reg занимает 4 байта, тогда как структурной переменной понадобится 7 байтов).

Структуры и объединения можно вкладывать друг в друга.

3. ВОПРОСЫ ДЛЯ САМОПРОВЕРКИ

1. Что такое структура и чем она полезна программисту?
2. Приведите пример, когда структурная переменная повышает эффективность разработки.
3. Чем объединение отличается от структуры? Приведите примеры, когда использование объединения более оправдано, чем структуры и наоборот.

4. САМОСТОЯТЕЛЬНАЯ РАБОТА

1. Заметив, что в задаче 1 переменные Stud_1, Stud_2, Stud_3 описаны подряд, оптимизируйте процесс копирования их в массив.
2. Опишите с помощью MASM структуру рабочего. В качестве атрибутов рассмотрите его возраст, фамилию, имя, отчество, стаж. Перепишите структуру, сформировав фамилию, имя и отчество одним полем структурного типа.
3. Требуется описать абстрактную модель процессора, в которой имеется 256 32-битных регистров одинаковой структуры (по типу EAX). Каждый регистр определяется номером (от 0 до 255). Опишите тип данных, реализующий подобную модель естественным образом. (Предполагается, что разные переменные – суть разные регистры).

ЗАКЛЮЧЕНИЕ

За последние годы программное и аппаратное обеспечение претерпело существенные изменения. Языки высокого уровня и современные программные платформы позволяют решать широкие классы задач, предоставляют готовые модули для конструирования собственных приложений.

Однако, несмотря на стремительное развитие, основные типы процессоров придерживаются в своей основе концепции архитектуры x8086. Кроме того, всегда будут существовать задачи, требующие максимальной оптимизации затрачиваемых ресурсов. Разработка эффективных алгоритмов является искусством вне зависимости от выбранного разработчиком инструмента.

Язык ассемблер поможет начинающему программисту понять тонкости своей работы, а профессионалу – достичь отличных результатов при решении специфических задач.

ПРИЛОЖЕНИЯ

Приложение 1

УСТАНОВКА И РАБОТА С MASM. ВИРТУАЛЬНАЯ DOS-МАШИНА

Для обучения курсу ассемблера понадобится следующая сборка (проконсультируйтесь с преподавателем):

- пакет ассемблера (две версии в папке MASM);
- эмулятор DOSBox;
- файловый менеджер Total Commander 5.50;
- отладчик Turbo Debugger.

Современные 64-битные операционные системы способны работать с 32-рязрядными и 64-разрядными приложениями, однако не позволяют работать с 16-битными[8]. В частности, это не позволит напрямую запустить 16-битную программу на ассемблере (а именно с таких и начато обучение в данном практикуме). Для решения подобной проблемы необходимы программы эмуляторы.

Виртуальная машина. Эмулятор

Виртуальная машина – программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы под управлением другой платформы.

Виртуальная машина позволяет запускать из одной операционной системы другую (например, Linux под Windows либо наоборот), эмулировать работу отдельных компонент и даже имитировать работу нескольких серверов на одном ПК. Цели могут быть различными: от учебно-исследовательских до моделирования или тестирования нового ПО на старом.

Необходимо учитывать, что выполнение программ в виртуальном режиме может проходить медленнее, чем на «родной» платформе, поскольку ЦП параллельно работает с основной операционной системой и виртуальной машиной.

В нашем случае необходим запуск 16-битного приложения. Здесь не требуется вызова виртуальной машины MS-DOS с полным

---

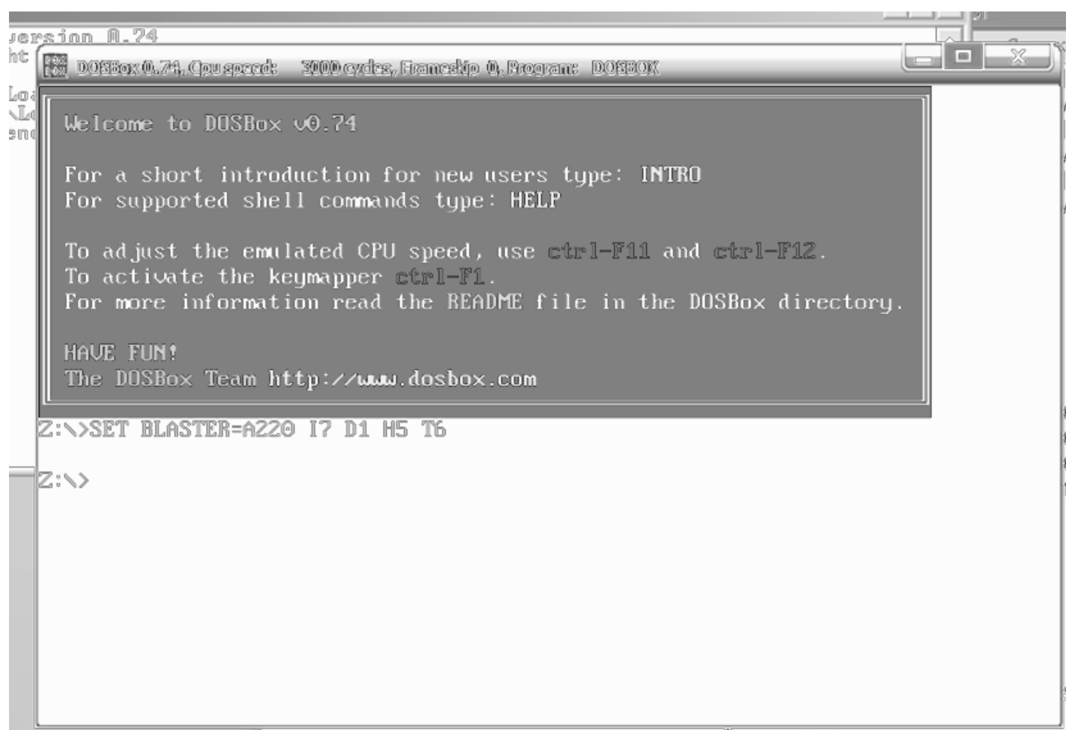
[8] Впрочем, не всегда и 32-битные ОС (например, Windows 7 x86) могут работать с такими программами. Это может быть связано и с более тонкой зависимостью от архитектуры материнской платы.

набором функций этой операционной системы, достаточно эмулировать только определённую их часть. Для этих целей хорошо подходит эмулятор DOSBox.

Эмулятор DOSBox. Установка и настройка

Перед установкой необходимо узнать разрядность вашей операционной системы и скачать соответствующую версию эмулятора.

Запустите программу:



DOSBox работает с виртуальным диском. Это реальный жесткий диск или любая выбранная директория на диске, которая рассматривается в качестве диска.

По умолчанию мы находимся в виртуальном диске Z. Введите команду help или help /all, чтобы просмотреть список доступных команд. Команда exit закрывает DOSBox.

Работа с программой похожа на работу с командной строкой. Для удобства навигации запустим файловый менеджер Norton Commander. Сначала смонтируем свой виртуальный диск (назовем его «С»), который фактически будет ссылаться на физический С. Команда для монтирования образа диска

mount C C:\

где C – буква создаваемого диска (любая, кроме Z). На самом деле под виртуальным диском можно брать некоторую директорию, например,

mount C C:\DOS

создает виртуальный диск с именем C, ссылающийся на папку DOS на диске C.

Вторая команда – выбор виртуального диска

C:

Пусть Norton Commander расположен по адресу C:\DOS\NC_5.50. Для запуска программы поочередно вводим команды

CD C:\DOS\NC_5.50

NC.exe

DOSBox можно настроить на автоматический запуск Norton Commander. Для этого откройте файл конфигурации DOSBox-0.74 > Options > DOSBox 0.74 options, найдите в документе раздел загрузки [autoexec] и допишите команды:

MOUNT C C:\

C:

CD C:\DOS\NC_5.50

NC.exe

Учитывайте, что Norton Commander – 16-битное приложение, поэтому обладатели 64-битной ОС могут запустить его только через DOSBox.

MASM 6.11 и MASM32

MASM 6.11 – более ранняя версия, рассчитанная на разработку приложений только для ОС MS-DOS.

MASM32 (на момент издания практикума последняя 11-я версия ассемблера) позволяет разрабатывать приложения как для MS-DOS, так и для Windows. Соответственно MASM32 более предпочтительный, универсальный.

Однако если операционная система не поддерживает работу с 16-битными приложениями, то запустить такую программу невозможно. Может показаться, что DOSBox способен решить эту пробле-

му. Но есть один момент – транслятор у MASM32 (ML.exe) и компоновщик (LINK16.exe) – 32-битные программы. Соответственно скомпилировать программу и получить 16-битный исполняемый файл можно даже напрямую без DOSBox[9]. Однако для ее запуска и отладки уже потребуется эмулятор.

Другое решение – установка MASM 6.11. Здесь ассемблер ML.exe и компоновщик LINK.exe уже полностью 16-битные. Установка ассемблера производится в режиме DOSBox.

Установка MASM 6.11

1. Разархивируйте файл во временную папку на диске C. В архиве будет 5 папок с программой (DISK1, DISK2, …), папка с документами и патч (последние можно удалить).
2. Запустите DOSBox, перейдите в каталог DISK1 и запустите инсталлятор setup.exe.
3. Изначально выбирайте первый (полный) тип установки, далее ориентацию под обе ОС: MS-DOS/Windows; следующий ряд вопросов помечайте как «NO». После этого можно подправить пути установки папок и нажать Enter для установки (имена пути рекомендуется писать на латинице и не более 8 символов). По окончании процесса выдается сообщение об успешной установке и дополнительная информация.
4. Теперь можно удалить папки DISK1, DISK2 и др. Ассемблер готов для работы.

Компиляция программы

Сформируйте папку для проекта. Поместите туда .asm-файл с кодом программы. Его можно получить через Norton Commander (shift + F4) или отдельно вне DOSBox. Также создается .bat-файл для компиляции. После чего через DOSBox запустите его. В случае успешной компиляции создается исполняемый файл программы (.exe или .com); дополнительные файлы (.obj, .map) можно удалить.

В качестве проверки скомпилируйте следующую программу program.asm:

```
.8086
.model small
```

[9] Здесь также возможны исключения либо компиляция произойдет с ошибкой.

```
.stack 100h

.data
Press   BYTE  "Press any key...$"

.code
Start:
        mov   ax,@data
        mov   ds,ax

        mov   ah,09h
        mov   dx,OFFSET Press
        int   21h

        mov   ah,08h
        int   21h

        mov   ah,4Ch
        int   21h

end    Start
```

Код для компилирующего .bat-файла:

```
@echo off
C:\MASM611\BIN\ML.EXE   /c  program.asm
C:\MASM611\BIN\LINK.EXE    program.obj,,,,,
PAUSE
```

Установка MASM32

Установка этого пакета еще проще: инсталлятор сам выполняет всю работу, периодически информируя о процессе установки.

Если операционная система еще поддерживает работу с 16-битными приложениями, то проблем возникнуть не должно и этот пакет можно использовать в качестве основного (для разработки 16- и 32-битных приложений).

И поскольку MASM32 разрабатывался с ориентацией на Windows, то он позволяет создавать уже привычные пользователю консольные и оконные приложения на базе библиотек функций Windows API.

Приложение 2

РАБОТА С ОТЛАДЧИКОМ. TURBO DEBUGGER

Для эффективного изучения ассемблера и поиска ошибок важно развить умение работы в отладчике.

Поскольку реальный и защищенный режимы работы процессора имеют существенные отличия в схеме адресации, то и отладчики ориентированы на каждый из них. Среди широкого спектра отладчиков в работе будут рассмотрены 16-битный отладчик от компании Borland Turbo Debugger и свободно распространяемый 32-битный OllyDbg.

Turbo Debugger

Turbo Debugger (TD) – один из наиболее мощных для своего времени отладчиков, предназначенный для дизассемблирования и отладки программного кода. TD предназначался для .exe и .com приложений в операционной системе MS-DOS. Впрочем, в TASM 5 был добавлен и отладчик для 32-битной архитектуры.

Если вы установили TASM 5, то отладчик находится в следующей директории:

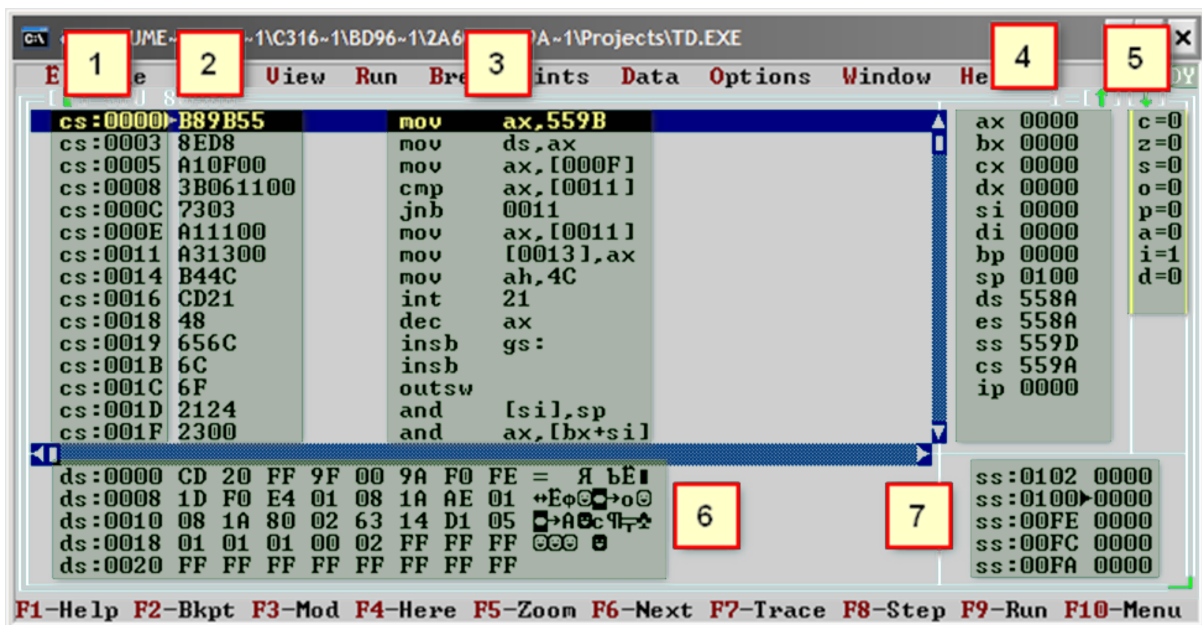
C:\tasm\bin\td.exe

После запуска появится окно отладчика. Для изучения основных возможностей отладчика будет использована следующая программа:

```
.8086
.model small
.stack 100h

.data
Mess    byte    "Hello!$"
n       word    23h
m       word    29h
max     word    ?

.code
Start:

    mov   ax,@data
    mov   ds,ax
```

```
        mov     ax,n
        cmp     ax,m
        jae     l1
        mov     ax,m
l1:
        mov     max,ax

        mov     ah,4Ch
        int     21h

end     Start
```

Требуется скомпилировать указанный код и получить .exe файл программы. Далее запустите отладчик и откройте программу. Рассмотрим основные блоки отладчика.



1. Адресное пространство программного кода. За сегмент кода отвечает регистр CS.
2. Опкоды команд в оперативной памяти.
3. Инструкции процессору.
4. Значение регистров в текущий момент времени.
5. Значение бит у регистра флагов.

6. Дамп памяти. Левая колонка задает адрес. Центральная выводит содержимое байт памяти (в строке по 8 байт). Правая содержит символы опкода с точки зрения ASCII представления.
7. Содержимое стека.

Из первого и второго столбцов можно узнать, сколько байт памяти занимает определенная команда: достаточно найти разность между адресами или посчитать количество байт в опкоде. Черная полоса является курсором.

Небольшой треугольник между первым и вторым столбцом – указатель команды, готовой к выполнению. Известно, что адрес выполняемой команды хранится в регистре IP, а весь адрес задается парой CS:IP. В нашем примере IP = 0 (что подтверждается колонкой 4).

Сразу обратим внимание, что код отображается не в точности, как в нашей программе. Как уже говорилось ранее, директивы не являются командами. Поэтому код начинается со строчки

```
mov    ax,@data
```

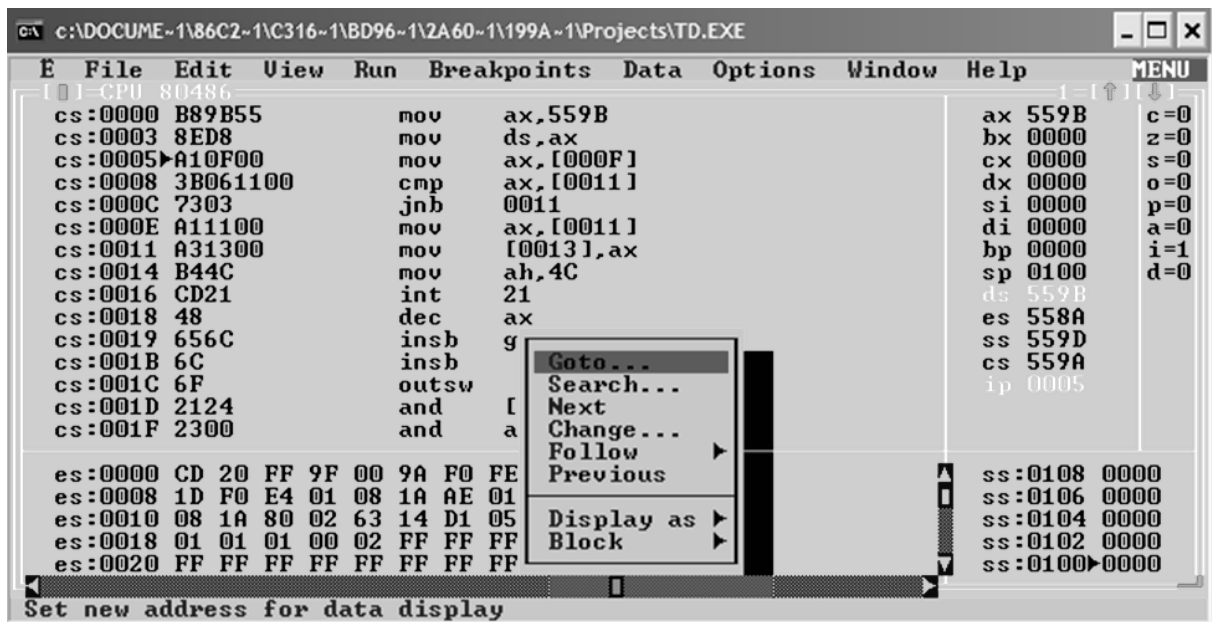
Пошаговое выполнение (трассировку) можно выполнять на различных уровнях.

- [F7] – выполняется одна команда;
- [F8] – выполняется одна команда, причем процедуры и циклы считаются одной командой;
- [F4] – выполнять до ближайшей контрольной точки (для выставления контрольной точки переведите курсор на команду и нажмите F2);
- [F9] – выполнить всю программу до конца.

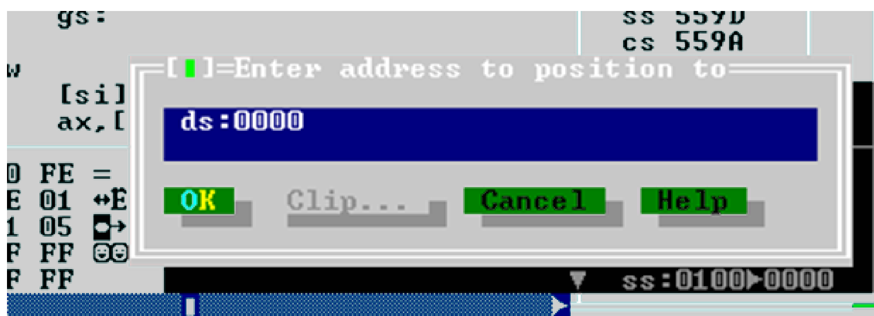
В первой команде вместо @data стоит значение 559B. Этот адрес определяется автоматически и далее сохраняется в регистр DS (адрес сегмента данных). Выполните первые две команды и посмотрите, что значения регистров AX и DS изменились.

Третья команда вместо переменной n пишет адрес этой переменной. Как известно, ассемблер обращается к памяти по адресу, поэтому такой вывод неудивителен.

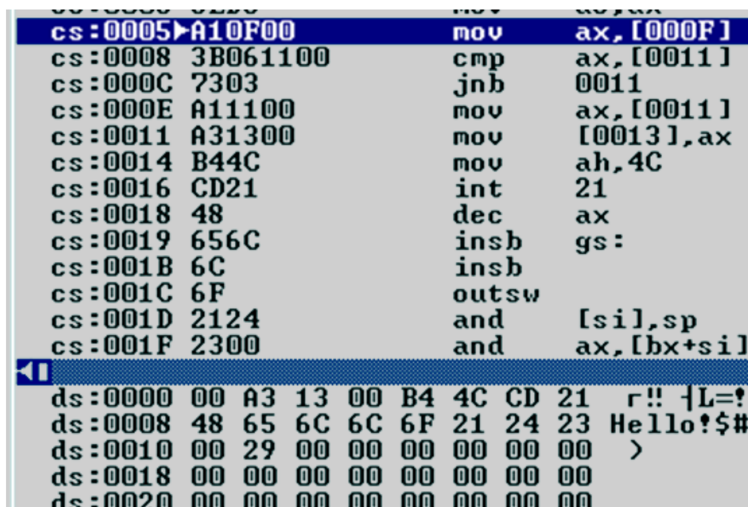
Посмотрим, как записаны переменные в памяти. Сегмент данных имеет адрес DS:0000, значение регистра DS уже задано. В нашем случае отображается другая область памяти. Чтобы перейти к требуемому участку, необходимо выделить окошко дампа, вызвать подменю и вкладку GOTO:



В появившемся окне вводим адрес перехода



Теперь отображается дамп памяти



Из примера видно, что переменная n имеет адрес 000F, а переменная m – 0011. Действительно, в дампе памяти по указанным адресам размещаются значения этих переменных

```
ds:0000 00 A3 13 00 B4 4C CD 21  r‼ ┤L=!
ds:0008 48 65 6C 6C 6F 21 24 23 Hello!$#
ds:0010 00 29 00 00 00 00 00 00  )
ds:0018 00 00 00 00 00 00 00 00
ds:0020 00 00 00 00 00 00 00 00
```

Третья переменная Mess-строка. Каждый символ занимает один байт, т. е. на хранение требуется 7 байтов. Правая колонка памяти отображает ASCII представление, т. е. каждый байт отображается как символ

```
0000 00 A3 13 00 B4 4C CD 21  r‼ ┤L=!
0008 48 65 6C 6C 6F 21 24 23 Hello!$#
0010 00 29 00 00 00 00 00 00  )
0018 00 00 00 00 00 00 00 00
0020 00 00 00 00 00 00 00 00
```

При выполнении команды cmp меняется значение регистра флагов, что можно проследить по колонке 5. По завершению отладки TD предлагает начать процесс заново.

Замечание. Для отладки можно использовать дополнительную информацию, генерируемую при компиляции. Для этого транслятору добавляется опция /Zi, а компоновщику /DEBUG. Это позволит, в частности, вместо адресов видеть имена переменных и меток.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Гордеева, И. А.* Ассемблер. Семинарские и практические занятия по курсу «Архитектура компьютера и основы микроэлектроники» / И. А. Гордеева, Е. П. Давлетярова, А. В. Шутов. – Владимир : Изд-во ВГГУ, 2010. – 47 с.
2. *Ирвин, Кип.* Язык ассемблера для процессоров Intel: пер. с англ. / Кип Ирвин. – 4-е изд., стер. – М. : Вильямс, 2005. – 912 с. – ISBN 5-8459-0779-9.
3. *Магда, Ю. С.* Ассемблер. Разработка и оптимизация Windows приложений / Ю. С. Магда. – СПб. : БХВ-Петербург, 2003. – 544 с. – ISBN 5-94157-324-3.
4. *Он же.* Ассемблер для процессоров Intel Pentium / Ю. С. Магда. – СПб. : Питер, 2006. – 410 с. – ISBN 5-469-00662-X.
5. *Митницкий, В. Я.* Архитектура IBM PC и язык Ассемблера : учеб. пособие / В. Я. Митницкий. – М. : Изд-во МФТИ, 2005. – 148 с. – ISBN 5-7417-0136-1.
6. *Юров, В. И.* Assembler : учеб. для вузов / В. И. Юров. – СПб. : Питер, 2003. – 637 с. – ISBN 5-94723-581-1.
7. *Он же.* Assembler : практикум / В. И. Юров. – 2-е изд., стер. – СПб. : Питер, 2006. – 399 с. – ISBN 5-94723-671-0.
8. Programmer’s Guide. Microsoft® MASM. Assembly-Language Development System Version 6.1. For MS-DOS® and Windows™ Operating Systems [Электронный ресурс]. – URL: http://people.sju.edu/~ggrevera/arch/references/MASM61PROGUIDE.pdf (дата обращения: 20.11.2016).
9. Сайт поддержки пакета MASM32 [Электронный ресурс]. – URL: http://www.masm32.com/ (дата обращения: 20.11.2016).

Учебное издание

ЯКУБОВИЧ Денис Андреевич
МЕДВЕДЕВ Юрий Алексеевич

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ АССЕМБЛЕР.
MACRO ASSEMBLER

Практикум

Редактор А. П. Володина

Технический редактор С. Ш. Абдуллаева

Корректор Е. В. Невская

Компьютерная верстка Л. В. Макаровой

Подписано в печать 17.05.17.
Формат 60×84/16. Усл. печ. л. 11,16. Тираж 60 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.