

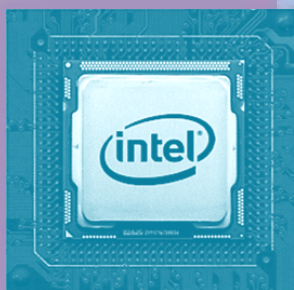
ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

Кафедра информационных систем и технологий

О.Л. Куляс, К.А. Никитин

Программирование на языке **ASSEMBLER**

Лабораторный практикум
по дисциплине
«ЭВМ и периферийные устройства»
(часть 1)



Самара
2016

УДК 004.43 (076)

К 907

Рекомендовано к изданию методическим советом ПГУТИ,
протокол № 8, от 14.04.2016 г.

Куляс, О.Л.

К 907 Программирование на языке ASSEMBLER: лабораторный практикум по дисциплине «ЭВМ и периферийные устройства» (часть 1) / О.Л. Куляс, К.А. Никитин. – Самара: ПГУТИ, 2016. – 87 с.

Лабораторный практикум предназначен для бакалавров направления 09.03.01 – «Информатика и вычислительная техника», изучающих курс «ЭВМ и периферийные устройства». Двухсеместровый цикл лабораторных работ включает 12 работ (7 работ в 1-й части и 5 работ во 2-й), которые позволяют освоить основы программирования на языке ASSEMBLER. Каждая лабораторная работа содержит достаточный теоретический материал, поэтапно вводящий студентов в мир программирования на языке Ассемблера, сведения и задания, необходимые для практического выполнения работы, список литературы, рекомендуемой для дополнительного изучения, а также контрольные вопросы для проверки усвоения изученного.

Лабораторный практикум можно использовать не только студентам, указанного направления подготовки, но и всем желающим самостоятельно овладеть основами программирования на языке ASSEMBLER.

©, Куляс О.Л., 2016

Оглавление

Введение	4
Лабораторная работа №1. Введение в основы программирования на языке Ассемблера	5
Лабораторная работа №2. Упрощённое оформление программ. Создание исполняемых *.com файлов	19
Лабораторная работа №3. Изучение команд передачи данных. Основы работы с отладчиком	27
Лабораторная работа №4. Программирование арифметических операций. Изучение основ работы с TURBODEBUGGER	40
Лабораторная работа №5. Исследование способов адресации операндов	49
Лабораторная работа №6. Работа с подпрограммами и процедурами ..	58
Лабораторная работа №7. Исследование организации переходов и циклов	74
Краткая система команд микропроцессора i80X86	86

Введение

Язык машинных кодов, в котором каждая машинная команда представляется последовательностью микроопераций процессора, является самым низкоуровневым и малокомфортным для программистов. Пришедший ему на смену язык Ассемблера (Assembly language), появившийся в 50-е годы 20 века, является следующим по уровню языком, в котором малопонятные двоичные или шестнадцатеричные кодовые комбинации (машинные коды) были заменены мнемоническими обозначениями (мнемониками) машинных команд. Каждая инструкция (команда) языка Ассемблера, как правило, соответствует одной машинной команде. Считается, что название языка произошло от названия программы-транслятора – **Assembler** (рус. сборщик), который производил автоматическое формирование машинных кодов из инструкций, представленных в виде мнемоник.

Программирование на языке Ассемблера невозможно без знания архитектуры микропроцессора и вычислительной машины в целом. Поэтому, программирование на языке Ассемблера является наилучшим инструментом для изучения базовых вопросов архитектуры и организации вычислительных машин.

Различные типы микропроцессоров используют свой диалект Ассемблера. Настоящий лабораторный практикум посвящен изучению языка Ассемблера для микропроцессоров, совместимых с базовым процессором **фирмы Intel i8086** и объединенных обозначением **i80X86**. В связи с ограниченным объемом дисциплины рассматривается программирование только в реальном режиме работы микропроцессора.

Несмотря на то, что современные языки программирования высокого уровня обеспечивают не только удобное, но и эффективное системное программирование, в тех случаях, когда особенно важно получить оптимальный объектный код, целесообразно использовать Ассемблер. Ассемблер с самого своего появления являлся лучшим языком для программирования аппаратных устройств. Основное преимущество этого языка заключается в полном контроле над аппаратными устройствами, кроме того, он обеспечивает получение наиболее компактного и эффективного кода. Использование этого языка позволяет создавать быстродействующие программы, которые с максимальной эффективностью используют ресурсы компьютера.

Лабораторная работа №1

Введение в основы программирования на языке Ассемблера

1 Цель работы

Практическое овладение навыками составления простейших программ на языке Ассемблера и работы с программами TASM и TLINK.

2 Теоретический материал

Несмотря на то, что современные языки программирования высокого уровня обеспечивают не только удобное, но и эффективное системное программирование, в тех случаях, когда особенно важно получить оптимальный объектный код, целесообразно использовать Ассемблер. Ассемблер представляет собой машинный язык в символической форме, которая достаточно понятна и удобна программисту.

2.1 Этапы создания программы на Ассемблере

Полный цикл создания программы на Ассемблере можно представить в виде последовательности четырех этапов, показанных на рис. 1.1.

Исходный модуль программы создается в любом текстовом редакторе, например, в «Блокноте» и сохраняется в виде файла с именем, присвоенным по правилам **MS DOS**, с обязательным расширением **asm**. Для нашей первой программы это будет **hello_1.asm**.

Для получения исполняемого модуля, который можно запустить на выполнение, требуется последовательно выполнить этапы **трансляции** и **компоновки**. Для этого используются программы, входящие в состав пакета Ассемблера **Turbo Assembler (TASM)** фирмы **Borland**.

Трансляция производится с помощью **компилятора**¹ Турбо Ассемблера, который является исполняемой программой **tasm.exe**, работающей в режиме командной строки. Он вызывается командой **DOS**:

tasm /z/zi/n <имя файла><имя файла><имя файла>,

где **/z** – ключ, разрешающий вывод на экран строк исходного текста программы, в которых Ассемблер обнаружил ошибки;

/zi – ключ, управляющий включением в результирующий файл полных сведений о номерах строк и именах исходного модуля;

¹Компилятор транслирует **весь текст** исходного модуля в ходе одного непрерывного процесса. При этом создается объектный модуль, который далее обрабатывается без участия компилятора.

/n – ключ, который исключает из листинга информацию о символических обозначениях в программе.

Следующие далее **имена трех файлов** – исходного (***.asm**), объектного (***.obj**) и листинга (***.lst**) можно использовать без расширений, например

tasm /z /zi /n hello_1 hello_1 hello_1



Рис. 1.1 – Этапы создания ассемблерной программы

Если исходный модуль не содержит ошибок, то на экран выводится сообщение об успешной трансляции, а в текущем каталоге появятся новые файлы – объектный (**hello_1.obj**) и листинга (**hello_1.lst**).

Компоновка объектных модулей с библиотечными модулями производится вызовом компоновщика **tlink.exe** из командной строки:

tlink/v <имя файла>

Ключ **/v** передает в загрузочный файл информацию, используемую при отладке программ. Следующее далее **имя файла** обозначает имя объектного модуля. Расширение в этом имени можно не указывать.

Для нашего примера компоновка будет осуществляться следующей командой:

tlink/v hello_1

В случае успешного окончания компоновки в текущем каталоге появляется исполняемый файл – загрузочный модуль **hello_1.exe**, и файл карты сборки **hello_1.map**.

Загрузочный модуль может быть запущен на выполнение командой **DOS**

hello_1.exe.

Для отладки создаваемых программ используется программа-отладчик **Turbo Debugger (td.exe)** фирмы **Borland**.

2.2 Структура исходного модуля

Исходный модуль программы на Ассемблере – последовательность строк, содержащих **командные операторы** и **директивы**, представляемая как функциональная единица для дальнейшей обработки. Исходный модуль создается текстовым редактором и хранится в виде текстового файла с расширением ***.asm**.

Алфавит допустимых символов Ассемблера включает в себя прописные и строчные буквы английского алфавита, арабские цифры и некоторые специальные символы, которые в Ассемблере имеют особый смысл. Кроме того, допускается использование некоторых непечатаемых символов, управляющих работой ЭВМ: **пробел (20h)**, **перевод строки (0Ah)**, **возврат каретки (0Dh)** и **табуляция (09h)**.

Прописные и строчные буквы Ассемблером не различаются, за исключением символьных констант.

Командные операторы или просто **команды** имеют следующий формат:

[метка:] [префикс] мнемоника [операнд(ы)] [:комментарий],

где **метка** – определяемое пользователем имя команды, заканчивающееся двоеточием. Значением метки является адрес отмеченной команды. Используются для организации команд передачи управления. Метка состоит из последовательности **символов** или **цифр**, однако всегда начинается с символов **английского алфавита** или с символов @, _, ?;

префикс – элемент, который служит для изменения стандартного действия команды;

мнемоника – мнемоническое обозначение команды, которое представляет собой ключевые слова Ассемблера и идентифицирует выполняемую командой операцию. Обычно используются сокращенные английские слова, передающие смысл команды;

операнд(ы) – объекты, которые участвуют в указанной операции. Это могут быть адреса данных или непосредственно сами данные, необходимые для выполнения команды. Команды могут быть двух, одно и безоперандными. Если операндов два, то они разделяются запятой;

комментарий – начинается с точки с запятой и предназначен для пояснения к программе. Может выполняться на русском языке, так как не влияет на выполнение программы.

Метка, префикс, мнемоника и операнд(ы) разделяются по крайней мере одним пробелом друг от друга.

Каждая команда при трансляции генерирует машинный код команды, размер которого зависит от способов задания операндов.

Директивы Ассемблера позволяют управлять процессом ассемблирования и формирования листинга. Их используют для распределения памяти, обеспечения связи между программными модулями и работы с символическими именами. Часто их называют **псевдокомандами**. Формат директив похож на формат команд:

[имя] директива [операнд(ы)] [;комментарий],

где **имя** – имя директивы. Никогда не заканчивается двоеточием и имеет совершенно другой смысл по сравнению с меткой. Некоторые директивы обязательно должны иметь имя, у некоторых оно может отсутствовать;

директива – аналогична полю мнемоники команды и содержит одно из ключевых слов Ассемблера, обозначающее название директивы;

операнд(ы) – аналогичны операндам команд и конкретизируют действия, выполняемые по данной директиве;

комментарий – пояснения к директиве.

В отличие от команд директивы действуют только в процессе ассемблирования программы и не генерируют машинных кодов.

Исходный модуль, как правило, состоит из нескольких фрагментов программы, сгруппированных по функциональным признакам – **логических сегментов**. Каждый из сегментов начинается с директивы **SEGMENT** (начало сегмента), которая обязательно должна иметь уникальное имя, и заканчивается директивой **ENDS** (конец сегмента) *с тем же именем*. Имена сегментов, как правило, несут смысловую нагрузку, ассоциируясь с назначением сегмента программы:

имя SEGMENT [параметры]

.....

имя ENDS

В **сегменте данных** программы определяются данные (переменные, символьные цепочки, массивы и др.) с которыми будет работать программа. Для определения данных чаще всего используются директивы **DB (Define Byte)** (определить байт), **DW (Define Word)** (определить слово), **DD (Define Doubleword)** (определить двойное слово), реже **DQ (Define Quadword)** (определить четыре слова) и **DT (Define Tenbyte)** (определить десять байт). **Директивы определения данных** имеют следующий формат:

$$[\text{имя переменной}] \left\{ \begin{array}{l} \text{DB} \\ \text{DW} \\ \text{DD} \end{array} \right\} <\text{нач. знач.}> [, <\text{нач. знач.}>,] \dots$$

В них определяется: сколько единиц памяти (байт) необходимо резервировать для переменной с указанным именем и как их инициализировать, если заданы начальные значения. В поле операнда требуется хотя бы одно начальное значение, но допустим и список начальных значений, элементы которого разделяются запятыми. Примерный вид типового сегмента данных представлен ниже:

Data	SEGMENT	;начало сегмента с именем Data
var_1	DB 11000110b	;определить переменную var_1 размером ;байт с начальным значением 11000110b
var_2	DW 9FFEH	;определить переменную var_2 размером ;слово с начальным значением 9FFEH
var_3	DW ?	;определить переменную var_3 размером ;слово не задавая ее начального значения
string	DB 'Assembler'	;определить строку символов string ;каждый символ которой имеет размер ;байт с начальным значением Assembler

```
mas_1 DB 0, -3, 28, 46, 39 ;определить массив с именем mas_1
                               ;состоящий из пяти числовых элементов
                               ;размером байт
Data ENDS                    ;конец сегмента с именем Data
```

Как видно из приведенного примера, для задания начальных значений могут использоваться числовые константы, представленные в двоичной (Binary - **b**), шестнадцатеричной (Hexadecimal - **h**), восьмеричной (Octal - **q**) или десятичной (Decimal - **d**) системах счисления. При этом за младшей цифрой числа должен следовать однобуквенный дескриптор системы счисления². **Если шестнадцатеричная константа начинается с буквы, то она обязательно должна быть дополнена слева незначащим нулем.**

После того как переменная объявлена в сегменте данных, каждая из них связывается с тремя атрибутами, которые используются программой:

сегмент (SEG) – идентифицирует сегмент, содержащий переменную;

смещение (OFFSET) – представляет собой расстояние в байтах от начала сегмента до переменной;

тип (TYPE) – идентифицирует единицу памяти, выделяемую для хранения переменной, т.е. байт, слово, двойное слово и т. д.

В **сегменте стека** программы резервируются ячейки памяти указанного размера для организации временного хранилища данных и результатов промежуточных вычислений при выполнении программы. Типичный сегмент стека приводится ниже:

```
Stk SEGMENT                 ;начало сегмента с именем Stk
DB 256 DUP (?)              ;отвести под стек 256 байт
Stk ENDS                    ;конец сегмента с именем Stk
```

Для резервирования ячеек памяти для стека используется директива **DB** с операндом **256 DUP (?)**. Эта конструкция **DUPLICATE** (повторять) имеет следующий формат:

n DUP (<нач. знач.> [, нач. знач., ...]),

здесь параметр **n** задает число повторений элементов, находящихся в круглых скобках.

² В обозначении десятичных чисел использование дескриптора не обязательно.

В **сегменте кода** программы приводится последовательность Ассемблерных команд, в соответствии с алгоритмом решаемой задачи. Типовое построение сегмента кода приводится ниже. Как и другие сегменты он начинается и заканчивается директивами **SEGMENT** и **ENDS** с именем Code. Однако при его написании следует придерживаться определенных правил:

- первая команда сегмента должна иметь **метку**, которая является точкой входа в программу;
- две первые команды сегмента инициализируют (загружают) **сегментный регистр данных DS** сегментным (базовым) адресом сегмента данных. Поскольку **сегментные регистры не допускают непосредственной загрузки**, она производится через **РОН**, в данном случае через **регистр-аккумулятор AX**;
- заканчиваться сегмент должен **корректным выходом в DOS**, что обеспечивается тремя последними командами.

Типичный вид сегмента кода показан ниже.

ASSUME DS:Data, SS:Stk, CS:Code ;назначить сегментные регистры

Code SEGMENT ;начало сегмента с именем Code

Start:

```

MOV AX, Data      ;загрузить сегментный регистр DS
MOV DS, AX        ;сегментным адресом сегмента данных
.....           ;здесь располагаются
.....           ;команды Ассемблера
.....           ;в соответствии с алгоритмом
.....           ;задачи
MOV AL, 0         ;завершить программу
MOV AH, 4Ch       ;с помощью
INT 21h           ;DOS

```

Code ENDS ;конец сегмента с именем Code

END Start ;конец исходного модуля

Перед первым использованием сегментных регистров и перед каждым местом в программе, где их содержимое может измениться, необходима директива **ASSUME** (предположить, считать). Целесообразно располагать ее перед сегментом кодов. Директива имеет следующий формат:

ASSUME <SR: базовый адрес>, [<SR: базовый адрес>], ... ,

здесь **SR** – сегментный регистр (**DS**, **SS**, **CS** или **ES**), а **базовый адрес** задает сегментный адрес области памяти, которая адресуется через этот сегментный регистр. Задать базовые адреса сегментов можно с помощью их имен, что и делается в приведенном выше примере.

Если **ASSUME DS:Data, SS:Stk, CS:Code**, то это означает, что базовый адрес сегмента данных с именем **Data** должен находиться в регистре **DS**, сегмента стека с именем **Stk** – в регистре **SS**, а сегмента кода с именем **Code** – в регистре **CS**. Тем не менее, эта директива не освобождает программиста от начальной инициализации сегментных регистров (см. две первые команды сегмента кодов).

Завершается исходный модуль директивой **END**, в качестве операнда которой используется точка входа в программу (**метка первой команды**). Тем самым сообщается Ассемблеру о достижении конца исходного модуля и дается указание начать выполнение программы с команды, помеченной меткой **Start**.

Порядок написания сегментов в исходном модуле значения не имеет, однако после загрузки программы в память, расположение сегментов будет таким же, как в исходном модуле.

2.3 Первая программа на Ассемблере

Исследуемая в данной работе программа **Hello_1**, позволяет вывести на экран монитора строку текста с именем **Greet– Hello, My friends!** (см. п.4.2).

Исходный модуль программы организован в виде трех сегментов, как было описано выше.

Данными является выводимая на экран строка, которая и объявляется в сегменте данных. Цифры **13** и **10** в строке объявления переменных являются управляющими символами и означают коды перевода строки и возврата каретки. Замыкающий символьную строку перепределенный символ **‘\$’** (доллар) это переменная, которая является счетчиком текущего адреса после определения символьной строки. Часто используется для вычисления длины символьной переменной.

В сегменте стека с именем **Ourstack** резервируются 256 ячеек памяти размером байт для организации стека.

В сегменте кода с именем **Code**, после инициализации сегментного регистра **DS**, следуют команды вывода строки символов на экран. Для этого:

- в регистр **АH** записывается номер функции вывода на экран **09h**;
- в регистр **DX** загружается начальный адрес выводимой строки символов **OFFSET Greet**. Длину выводимой строки можно не ука-

ывать, т.к. вывод будет происходить до символа доллара в конце строки;

- выполняется **команда прерывания int 21h**, вызывающая прерывание с типом **21h**, которое позволяет обратиться к служебным функциям **операционной системы MS DOS**.

В результате происходит обращение к **служебным функциям MS DOS**, из которых выбирается функция вывода строки символов и строка с указанным именем появляется на экране.

Аналогично производится завершение программы:

- в регистр **AL** заносится код успешного завершения программы **0**;
- в регистр **AH** записывается номер функции завершения **4Ch**;
- выполняется **команда прерывания** с типом **21h**.

В результате происходит обращение к **служебным функциям MS DOS** из которых выбирается функция завершения программы и программа корректно завершает свою работу.

2.4 Особенности работы с Ассемблером для MS DOS

На компьютерах с 64-х разрядными операционными системами Windows 7, 8, 10 не удастся запустить программы реального режима **MS DOS** из-за несовместимости. В этом случае используем **эмулятор системы DOS – DOSBox**. **DOSBox** является свободно распространяемой программой с открытым исходным кодом. Для работы через **DOSBox** нужно использовать некоторые простые приемы, описанные ниже.

Предположим, что Ваши файлы хранятся на диске **D:\UCHEBA\ASM**. Для работы с использованием **DOSBox** нужно выполнить следующее:

- запустить **DOSBox** с рабочего стола компьютера. На экране появятся два окна, одно из которых является окном состояния и служит для вывода служебных сообщений, а второе является рабочим. Окно состояния можно свернуть.

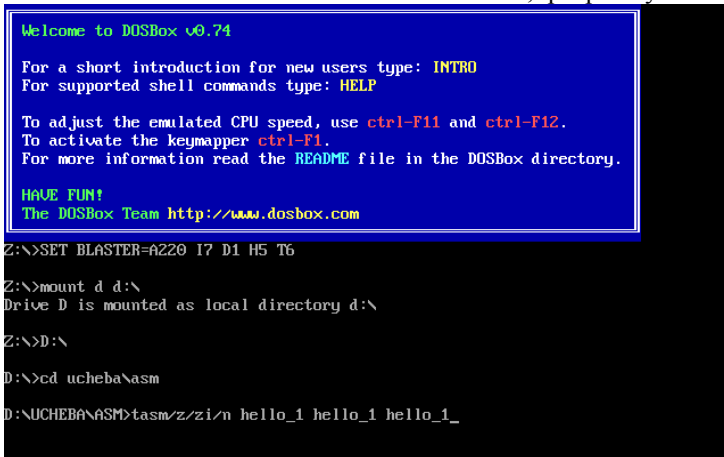
- создать локальный рабочий каталог с именем, совпадающим с именем диска, на котором находятся Ваши файлы. Для этого в командной строке рабочего окна **DOSBox** набираем:

```
z:\>mount D D:\
```

После сообщения о том, что рабочий каталог создан, работаем из командной строки **DOSBox** как из обычной командной строки **MSDOS**:

```
z:\>D:\ ; перейти в корневой каталог диска D;  
D:\>cd UCHEBA\ASM; перейти в папку ASM, сделав ее текущей;
```

D:\UCHEBA\ASM>tasm/z/zi/n hello_1 hello_1 hello_1; запустить
;программу tasm.exe



```
Welcome to DOSBox v0.74

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount d d:\
Drive D is mounted as local directory d:\

Z:\>D:\

D:\>cd ucheba\asm

D:\UCHEBA\ASM>tasm/z/zi/n hello_1 hello_1 hello_1_
```

Рис. 1.2 – Начальное окно DOSBox и отображаемые в нем команды

DOSBox обладает достаточно большим набором возможностей, которые можно узнать набрав в командной строке

z:>intro.

Для получения сведений о командах DOS можно набрать

z:>help.

Следующие комбинации нажатий клавиш позволяют управлять режимами **DOSBox**:

- ALT+Enter** – переключение полный/уменьшенный экран;
- CTRL+F1** – переназначение кнопок клавиатуры;
- CTRL+F5** – сделать снимок экрана в виде файла *.png;
- CTRL+F9** – закрыть программу;
- Escape** – закрыть графическое окно.

3 Подготовка к работе

3.1 Изучить методические указания и рекомендованную литературу.

3.2 Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

4.1 На диске **C** или **D** создать папку (каталог) с именем «**Family name**»³, например:

D: \USER \Petrov.

Скопировать в созданную папку файлы Турбо Ассемблера **tasm.exe**, **tlink.exe** и **td.exe** из папки **TASM**.

4.2 Используя текстовый редактор, создать и отредактировать исходный модуль программы **hello_1.asm**, текст которого приведен ниже.

```
;Program Hello_1– Ваша первая программа
Data SEGMENT                                ;Открыть сегмент данных
    Greet DB 'Hello, My friends!', 13, 10, '$' ;Определить строку
                                                ;символов с именем Greet
Data ENDS                                    ;Закрыть сегмент данных
Ourstack SEGMENT Stack                      ;Открыть сегмент стека
DB 100h DUP (?)                             ;Отвести под стек 256 байт
Ourstack ENDS                               ;Закрыть сегмент стека
ASSUME CS:Code, DS:Data, SS:Ourstack        ;Назначить сегментные
                                                ;регистры
Code SEGMENT                                ;Открыть сегмент кодов
Start:   mov AX, Data                        ;Инициализировать
         mov DS, AX                          ;сегментный регистр DS
         mov AH, 09h                         ;Вывести строку Greet
         mov DX, OFFSET Greet                ;на экран с помощью
         int 21h                             ;DOS
         mov AL, 0                           ;Завершить программу
         mov AH, 4Ch                         ;с помощью
         int 21h                             ;DOS
Code ENDS                                    ;Закрыть сегмент кодов
        END Start                           ;Конец исходного модуля
```

4.3 Используя компилятор Турбо Ассемблера **tasm.exe** создать файлы **hello_1.obj** и **hello_1.lst**. Просмотреть на экране тексты созданных файлов **hello_1.obj**, **hello_1.lst** и проанализировать их.

³При присвоении имен следует использовать правила MS DOS.

4.4 Используя компоновщик **tlink.exe** создать файлы **hello_1.exe** и **hello_1.map**. Вывести на экран файлы **hello_1.exe**, **hello_1.map** и проанализировать их.

4.5 Убедиться в работоспособности программы **hello_1**, запустив ее из командной строки.

4.6 Создать, для ускорения процесса ассемблирования и компоновки, **командный файл** с любым именем с расширением **bat (*.bat)**. Для этого в текстовом редакторе «Блокнот» наберите текст, состоящий из последовательности выполняемых команд **DOS** и сохраните его с расширением **bat**:

```
tasm /z /zi /n hello_1 hello_1 hello_1
tlink /v hello_1
```

Удалите из текущего каталога все файлы **hello_1**, кроме исходного, и проверьте работоспособность созданного командного файла, запустив его из командной строки.

4.7 Создать универсальный командный файл **run.bat**, который можно использовать для ассемблирования и компоновки любой создаваемой Вами программы. Для этого в командном файле, созданном в п. 4.6, имена файла следует заменить на символы **%1**:

```
tasm /z /zi /n %1 %1 %1
tlink /v %1
```

Для запуска универсального командного файла нужно в командной строке после **имени командного файла** (без расширения) указать имя Вашего **исходного модуля** без расширения, например:

```
run hello_1 ;запустить командный файл с именем run
;для исходного модуля hello_1.
```

Удалите из текущего каталога все файлы **hello_1**, кроме исходного, и проверьте работоспособность созданного командного файла, запустив его из командной строки.

4.8 Внести изменения в программу **hello_1**, которые заставят ее выводить на экран еще две строки символов, например, «**My name is Family name**» и «**My group IST-41**». Для этого создайте новый исходный модуль **hello_2.asm**, выполните ассемблирование и компоновку, после чего убедитесь в работоспособности программы.

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также Ф. И.О. студента, подготовившего отчёт;
- цель работы;
- листинги исходного модуля и всех файлов, созданных в процессе ассемблирования и компоновки для программ **hello_1** и **hello_2** с комментариями: ***.asm**, ***.obj**, ***.lst**, ***.map**, ***.exe**;
- листинги созданных командных файлов ***.bat** с комментариями.

6 Контрольные вопросы

- 6.1 Команды и директивы Ассемблера. Формат и отличия.
- 6.2 Какова цель сегментации памяти?
- 6.3 Что такое базовый адрес сегмента?
- 6.4 Какие значения может принимать базовый адрес сегмента?
- 6.5 Каков максимальный размер сегмента и почему?
- 6.6 Из каких логических сегментов состоит исходный модуль ассемблерной программы?
- 6.7 Какими директивами описывается сегмент?
- 6.8 Как описываются различные типы данных, используемые программой?
- 6.9 Каково назначение директивы ASSUME?
- 6.10 В чем заключается инициализация сегментных регистров?
- 6.11 Что такое ассемблирование и компоновка программы?
- 6.12 Что представляет собой исходный модуль программы?
- 6.13 Опишите стандартное начало и окончание сегмента кодов?
- 6.14 Каково содержание файлов с расширениями ***.ASM**, ***.LST**, ***.OBJ**, ***.MAP**, ***.EXE**?
- 6.15 Каково назначение и в чём отличия метки команды от имени директивы?
- 6.16 Для чего требуется пометить начальную команду программы меткой?
- 6.17 Как завершается исходный модуль программы?
- 6.18 Для чего предназначена программа DOSBox?
- 6.19 Как сделать Вашу рабочую папку текущей при использовании программы DOSBox?
- 6.20 Каким образом можно сохранить копию экрана в программе DOSBox?

7 Рекомендуемая литература

- 7.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с.121...141.
- 7.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 22...31.
- 7.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К.Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 25...35.

Лабораторная работа №2

Упрощённое оформление программ. Создание исполняемых *.com файлов

1 Цель работы

Практическое овладение навыками упрощённого оформления простейших программ на языке Ассемблера и работы с программами TASM и TLINK.

2 Теоретический материал

2.1 Упрощенное оформление программ

Для создания простых программ, которые содержат по одному сегменту для данных и кода, а также для программ, модули которых предполагается связывать с программами на языках высокого уровня, используются упрощенные директивы сегментации. При этом структура исходного модуля несколько упрощается.

Такие программы начинаются с директивы указания модели памяти **.MODEL**, которая в простейшем случае имеет формат:

.MODEL<модель памяти> [, язык],

где – **<модель памяти>** – имя выбранной модели памяти, является обязательным операндом;

– **[язык]** – имя языка программирования, на котором написаны вызываемые из данного модуля процедуры. Это может быть C, Pascal, Basic и другие.

Если программа написана полностью на Ассемблере, то достаточно указать только модель памяти. Возможные модели памяти и их описание приведены в таблице 2.1. Как видно из этой таблицы выбор директивы **.MODEL** определяет набор сегментов программы, размеры сегментов данных и кода, способы связывания сегментов и сегментных регистров. Директиву **ASSUME**, в отличие от способа традиционного оформления сегментов, не используют.

Для программ на Ассемблере используется модель памяти Small.

Применение директивы **.MODEL** обязывает программиста использовать для описания сегментов исходного модуля упрощенные директивы описания сегментов. Каждая из моделей может использовать директивы, представленные в таблице 2.2.

Таблица 2.1

Модели памяти

Имя	Тип кода	Тип данных	Определение	Назначение модели
Tiny	near	near	CS=DS=SS=dgroup Код и данные объединены в одну группу с именем dgroup .	Для создания *.com программ
Small	near	near	CS=_text; Код занимает один сег- мент. Данные объединены в одну группу с именем dgroup .	Для небольших и средних программ (*.exe). Для про- грамм на Ассемб- лере.
Medium	far	near	CS=<model>_text Несколько сегментов кода. Данные объединены в од- ну группу с именем dgroup .	Для больших про- грамм с малым объемом данных.
Compact	near	far	CS=_text Код в одном сегменте. Данные объединены в од- ну группу с именем dgroup .	Сегмент кода 64К. Объем данных не ограничен.
Large	far	far	CS=<model>_text Код в нескольких сегмен- тах. Данные объединены в одну группу с именем dgroup .	Размер кода и данных не ограни- чены. Для боль- ших программ.
Huge	far	far	CS=<model>_text	Тоже, что и large. Ведена для со- вместимости с ЯВУ.
Tchuge	far	far	CS=<model>_text	Используется при программирова- нии на TurboC и BorlandC++.
Flat	near	near	CS=text DS=SS=flat	Используется в среде OS/2.

Для упрощения ссылок к именам сегментов введены несколько предопределенных идентификаторов, которые могут использоваться в программе. Основные из них представлены в таблице 2.3.

Таблица 2.2

Упрощенные директивы описания сегментов

Директива	Описание
.code	Сегмента кода
.data	Сегмента инициализированных данных
.const	Сегмента постоянных данных
.data?	Сегмента неинициализированных данных
.stack [размер]	Сегмента стека. Параметр задает размер стека
.fardata [имя]	Сегмента инициализированных данных типа far
.fardata? [имя]	Сегмента неинициализированных данных типа far

Таблица 2.3

Идентификаторы, создаваемые директивой MODEL

Имя идентификатора	Значение: Физический адрес сегмента
@code	Кода
@data	Данных типа near
@data?	Неинициализированных данных типа near
@fardata	Данных типа far
@fardata?	Неинициализированных данных типа far
@stack	Стека

Для создания Ассемблерных программ с упрощенными директивами сегментации можно использовать приводимый ниже шаблон:

;Forma_1 – упрощенное оформление программ

```
.MODEL SMALL           ; модель памяти ближнего типа
.STACK 100h           ; определить стек размером 100h
.DATA                 ; открыть сегмент данных
```

В этом сегменте определяются данные с использованием символических имен

```
.CODE                 ; открыть сегмент кодов
Start:
    mov AX, @DATA     ; инициализировать
    mov DS, AX        ; сегментный регистр DS
```

Здесь следуют команды, которые определяются алгоритмом решаемой задачи

mov AL, 0	; завершить программу
mov AH, 4Ch	; с помощью
int 21h	; DOS
END Start	; конец исходного модуля.

2.2 Создание исполняемых модулей типа *.com

По внутренней организации все программы, написанные для MS DOS, принадлежат к одному из двух типов. Каждому из них соответствуют имена с расширениями ***.EXE** или ***.COM**. Программы, составленные в лабораторной работе №1 с помощью стандартных директив сегментации, относятся к наиболее распространенному типу **.EXE приложений**. Для таких программ характерно наличие отдельных сегментов данных, стека и команд. Для адресации к этим сегментам используются свои сегментные адреса. Такие программы удобно расширять за счет увеличения числа сегментов. Каждый сегмент в памяти может занимать размер **64 Кбайта**, однако в сумме этот объем может быть значительно увеличен.

Во многих случаях объем программы оказывается значительно меньше **64 Кбайт**. Такую программу нет никакой необходимости составлять из нескольких сегментов. В этом случае и данные, и стек, и команды можно разместить в одном сегменте, настроив все сегментные регистры на его начало. Исполняемые файлы, составленные по этим правилам, имеют расширение ***.COM**. В виде **COM** приложений обычно пишутся резидентные программы и драйверы, хотя в таком виде можно оформить любую прикладную программу.

Для написания исходных текстов программы типа **COM** можно использовать приведенный ниже шаблон:

;Forma_2 – исходный модуль для создания *.com приложения	
.MODEL TINY	; модель памяти ближнего типа
.CODE	; открыть сегмент кодов
ORG 100h	; отвести 256 байт под PSP
Begin: jmp Start	; безусловный переход на
	; первую команду

Здесь определяются данные с использованием символических имен

Start:

Здесь следуют команды, которые определяются алгоритмом решаемой задачи

```

mov AL, 0           ; завершить программу
mov AH, 4Ch         ; с помощью
int 21h             ; DOS
END Begin           ; конец исходного модуля.
    
```

Из исходного текста видно, что программа использует упрощенные директивы сегментации и содержит один сегмент – **сегмент кода**. Оператор **ORG 100h** резервирует 256 байт в памяти для **сегмента префикса программы (Program Segment Prefixs – PSP)**. Этот сегмент содержит таблицы и поля данных, которые заполняются и используются системой в процессе выполнения программы.

Данные, необходимые программе, можно объявить перед командами, внутри них или в конце сегмента. Однако при загрузке программы типа **.COM** регистр счетчика команд **IP** всегда инициализируется числом **100h**, поэтому вслед за оператором **ORG 100h** должна стоять первая выполняемая программой команда. В нашем случае это команда безусловного перехода на метку **Start**:

Begin: jmp Start;

После загрузки программы в память все сегментные регистры указывают на начало сегмента, в котором располагается программа, фактически на начало **PSP**. Регистр указателя стека **SP** автоматически загружается числом – начальной точкой входа в стек **FFFEh**. Таким образом, независимо от размера программы, под нее отводится **64 Кбайта** адресного пространства. Всю нижнюю часть занимает стек, размер которого заранее не определен, а зависит от работы программы. Образ программы в памяти показан на рис. 2.1.

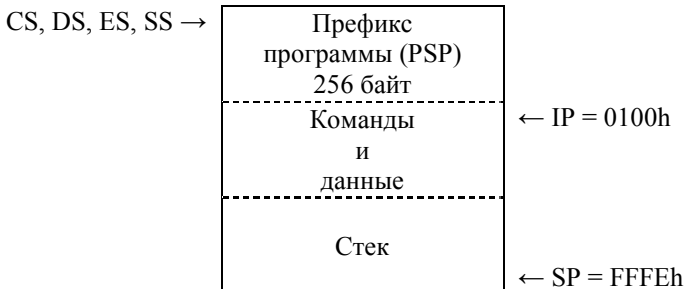


Рис. 2.1 – Образ программы **COM** в памяти

Из рисунка видно, что программа состоит из единственного сегмента, содержимое которого почти точно отражает содержимое исходного модуля. Отличие заключается в том, в исходном модуле отсутствует **префикс программы (PSP)**, который появляется в памяти в процессе загрузки программы.

Процесс ассемблирования исходного модуля происходит как обычно, а при вызове компоновщика **TLINK.EXE** в командной строке следует задавать **ключ /t**, который заставляет формировать исполняемый файл типа ***.COM**:

tlink /t <имяфайла>.

3 Подготовка к работе

- 3.1 Изучить методические указания и рекомендованную литературу.
- 3.2 Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

- 4.1 Используя текстовый редактор, создать и отредактировать исходный модуль программы **hello_2.asm**, (см. лаб. раб. №1) с использованием директив упрощенного оформления программ. Сохранить исходный текст на диске с именем **hello_2s.asm**. При наборе исходного текста используйте шаблон **Forma_1**, описанный в п.2.1.
- 4.2 Используя компилятор Турбо Ассемблера **tasm.exe** создать файлы **hello_2s.obj** и **hello_2s.lst**.
- 4.3 Используя компоновщик **tlink.exe** создать файлы **hello_2s.exe** и **hello_2s.map**.
- 4.4 Убедиться в работоспособности программы **hello_2s**.
- 4.5 Просмотреть в текстовом редакторе тексты всех созданных файлов и проанализировать их. Составить модель размещения сегментов программы (образ программы) **hello_2s.exe** в памяти ЭВМ.
- 4.6 Используя текстовый редактор, создать и отредактировать исходный модуль предыдущей программы для создания исполняемого файла типа ***.com**. Сохранить исходный текст на диске с именем **hello_2c.asm**. При наборе исходного текста используйте шаблон **Forma_2**, приведенный в п. 2.2.
- 4.7 Выполнить ассемблирование и компоновку созданного исходного модуля для создания исполняемого файла типа ***.com**.
- 4.8 Убедиться в работоспособность созданного исполняемого файла.

4.9 Просмотреть в редакторе все созданные файлы и проанализировать их. Составить модель размещения сегментов программы (образ программы) `hello_2c.com` в памяти ЭВМ.

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также фамилии И.О. студента, подготовившего отчёт;
- цель работы;
- листинги исходного модуля и всех файлов, созданных в процессе ассемблирования и компоновки для программ **hello_2s** и **hello_2c** с комментариями: ***.asm**, ***.obj**, ***.lst**, ***.map**, ***.exe**, ***.com**;
- список команд, использующихся для ассемблирования и компоновки;
- образы созданных программ **hello_2s.exe** и **hello_2c.com** в памяти.

6 Контрольные вопросы

- 6.1 Команды и директивы Ассемблера. Формат и отличия.
- 6.2 Какова цель сегментации памяти?
- 6.3 Сколько и каких сегментов может иметь программа?
- 6.4 В каких случаях имеет смысл использовать упрощенную сегментацию?
- 6.5 Какие директивы упрощенной сегментации используются?
- 6.6 Какие модели памяти используются при упрощенной сегментации?
- 6.7 Какими директивами описывается сегмент?
- 6.8 В чем заключается инициализация сегментных регистров и как она производится при упрощенной сегментации?
- 6.9 Как производится ассемблирование и компоновка программы при упрощенной сегментации?
- 6.10 Как выглядит типовая форма для создания **.exe приложений** упрощенной сегментацией?
- 6.11 Каким образом располагается программа типа **.exe** после ее загрузки в память?
- 6.12 В каких случаях используются программы типа **.com**?
- 6.13 Как выглядит типовая форма для создания **.com приложений**?
- 6.14 Из каких сегментов состоит исходный модуль программы типа **.com**?
- 6.15 Каким образом располагается программа типа **.com** после ее загрузки в память?
- 6.16 Каков размер области **сегмента префикса программы (PSP)**?
- 6.17 Каков размер стека в программах типа **.com**?

- 6.18 Как производится ассемблирование и компоновка программ типа **.com**?
- 6.19 Что представляет собой образ программы в памяти ЭВМ?
- 6.20 Что требуется для того, чтобы представить образ программы в памяти ЭВМ?

7 Рекомендуемая литература

- 5.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с.103...110.
- 5.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 96...107.
- 5.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К. Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 107...120.

Лабораторная работа №3

Изучение команд передачи данных.

Основы работы с отладчиком

1 Цель работы

Изучение команд передачи данных и практическое овладение навыками работы с отладчиком TURBODEBUGGER.

2 Теоретический материал

Современные процессоры поддерживают два основных режима работы – **реальный** и **защищенный**. **Реальный режим** (режим MSDOS) является однозадачным режимом с максимально возможной памятью 1 Мбайт. **Защищенный режим** является многозадачным и позволяет работать с памятью до 4 Гбайт.

Предметом нашего изучения является **реальный режим** работы процессора. Несмотря на то, что подавляющее число современных процессоров являются **32-х разрядными**, они начинают работу в **16-и разрядном реальном режиме**. Для перевода в 32-х разрядный реальный режим работы ассемблерная программа должна иметь в своем составе одну из директив **.386**, **.486**, **.586**. Эти директивы разрешают использование расширенных 32-х разрядных регистров и дополнительных команд Ассемблера, которые появлялись в его каждой новой модификации.

2.1 Программная модель вычислительной машины

Ассемблер учитывает все аппаратные и программные особенности устройств, которые участвуют в выполнении программы. Поэтому программирование на ассемблере требует знания аппаратных ресурсов, которыми располагает программист. Ресурсы, которые имеются в распоряжении программиста, принято показывать с помощью **программной модели** вычислительной машины. Она включает только те элементы, которые доступны на уровне команд Ассемблера. Как видно из рис. 3.1 в модели представлены программно доступные элементы 32-х разрядного микропроцессора, память и устройства ввода-вывода.

Несмотря на то, что современные процессоры насчитывают несколько десятков программно доступных регистров, в реальном режиме их число сокращается до **16**. Регистры принято объединять в четыре группы, как показано на рисунке. **Первую группу** составляют **регистры данных** или **регистры общего назначения (РОН)**. Это четыре 32-х разрядных регистра, предназначенных для хранения данных **EAX**, **EBX**, **ECX**, **EDX**. Они наиболее часто используются в арифметических

и логических операциях. Несмотря на то, что их разрядность 32 бита, их младшая половина может использоваться для хранения 16-и разрядных данных. Младшие половины этих регистров обозначаются как **AX, BX, CX, DX**. При работе с байтами младшие половины этих регистров могут быть поделены пополам на две восьмиразрядные части: **нижнюю - Low** и **верхнюю - High**. Для обращения к этим восьмиразрядным частям регистров используются обозначения **AH, AL; BH, BL; CH, CL; DH, DL**. В некоторых командах **POHы** используются со специальными функциями, поэтому иногда их используют с названиями, показанными на рисунке.

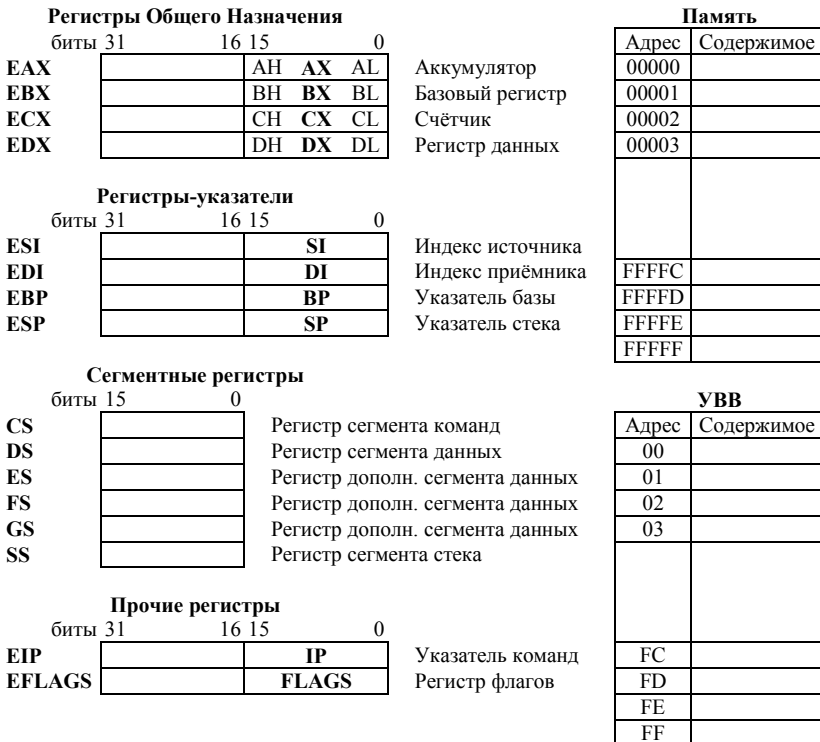


Рис. 3.1 – Программная модель 32-х разрядной ЭВМ в реальном режиме

Вторую группу составляют регистры указатели: два указательных регистра **ESP** и **EBP** и два индексных регистра **ESI** и **EDI**. Они предназначены для хранения 32-х или 16-и разрядных адресов.

Для хранения 16-и разрядных адресов используются младшие половины этих регистров **SP, BP, SI, DI**. Эти регистры также могут использоваться для выполнения арифметических и логических операций. **Указательные регистры SP, BP** предназначены для организации доступа к данным, находящимся в сегменте стека. **Индексные регистры SI, DI** предназначены для организации адресации к текущему сегменту данных. В некоторых командах эти регистры специфицированы, что отражается в их названиях.

Третья группа регистров содержит шесть 16-и разрядных сегментных регистров **CS, DS, ES, FS, GS, SS**. Они используются для хранения базовых (сегментных) адресов логических сегментов программы в памяти. По умолчанию в них хранятся сегментные адреса:

В **регистре CS** – текущего сегмента кода программы (Code Segment);

В **регистре DS** – текущего сегмента данных программы (Data Segment);

В **регистре SS** – текущего сегмента стека программы (Stack Segment);

В **регистрах ES, FS, GS** – дополнительных сегментов данных.

Четвертая группа состоит из двух регистров:

– **Регистр указателя команд EIP** используется только в 32-х разрядных приложениях и хранит смещение следующей подлежащей выполнению команды. В 16-и разрядных приложениях под MS DOS смещения могут быть только 16-и разрядными, поэтому используется только младшая часть указателя команд **IP**. Доступ к регистру **EIP (IP)** производится с помощью команд передачи управления.

– **Регистр флагов EFLAGS** (расширенный регистр флагов) также является 32-х разрядным. Однако задействовано в нем только 22 младших разряда, часть из которых используется только в защищенном режиме.

В **16-и разрядном реальном режиме** работы используются только младшие части расширенных регистров и четыре из шести сегментных – **CS, DS, ES, SS**.

Память, в рассматриваемой модели, представляет собой 8-и разрядные ячейки, каждая из которых характеризуется уникальным номером (адресом). Этот адрес называется **физическим** или **полным адресом PA**. Для реального режима работы физический адрес является **20-и битовым** и лежит в диапазоне:

в десятичной системе **0d ...2²⁰-1d**;

в шестнадцатеричной **00000h ...FFFFFFh**.

Такой диапазон адресов определяет **адресное пространство реального режима работы** и позволяет адресоваться к памяти емкостью $2^{20} = 1\text{Мбайт}$.

Устройства ввода-вывода (УВВ) подключаются к системе через коммутационные **порты ввода-вывода**. За каждым из внешних устройств закреплен один или группа адресов. В порт с нужным адресом можно **выводить** (записывать) информацию, либо **вводить** (читать) информацию из порта.

2.2 Команды передачи данных

Команды передачи данных (команды пересылок) предназначены для организации пересылки данных между регистрами, регистрами и памятью, памятью и регистрами, а также для загрузки регистров или ячеек памяти данными. При выполнении команд передачи данных **флаги не устанавливаются**.

Наиболее часто используемой командой передачи данных является команда **MOV**. Её формат следующий:

MOV dst, src ;dst:= (src).

Команда осуществляет **передачу содержимого источника (src) в получатель (dst)**. Операндами этой команды могут быть:

- регистр – регистр;
- регистр – память;
- память – регистр;
- регистр – непосредственные данные;
- память – непосредственные данные.

Команда обмена данными позволяет обменивать содержимое любого общего регистра и ячейки памяти, либо любой пары общих регистров:

XCHG op1, op2; op1:= (op2), op2:= (op1).

Здесь op1 и op2 первый и второй операнды команды.

Использование сегментных регистров в командах обмена запрещается.

Команда загрузки исполнительного адреса загружает в регистр **reg**, указанный в качестве первого операнда, относительный адрес второго операнда, который находится в памяти:

LEA reg, mem; reg:= [mem].

Не допускается использование сегментных регистров.

Команды работы со стеком используются для занесения данных в стек и извлечения данных из стека. Для адресации к вершине стека используется **регистр указателя стека SP**, который при выполнении стековых команд автоматически модифицируется. Все стековые команды манипулируют только двухбайтовыми данными – словами.

PUSH src; SP:= (SP) – 2 , [(SS):(SP)]:= (src).

Это команда **PUSH – поместить в стек**. Она уменьшает на 2 содержимое указателя стека **SP** и заносит на вершину стека по этому адресу двухбайтовый операнд, указанный в команде. В качестве операнда может использоваться любой 16 разрядный регистр или двухбайтовая ячейка памяти.

Команда извлечь из стека имеет формат

POP dst; dst:= [(SS):(SP)], SP:= (SP) + 2.

Команда извлекает 16-ти разрядные данные из ячеек стека, на которые указывает указатель **SP** и помещает их в получатель, указанный в команде. Содержимое **SP** при этом автоматически **увеличивается на 2**.

2.3 Отладчик Turbo Debugger

Отладчик позволяет отлаживать программы на уровне исходного текста. Предназначен для использования с Турбо языками фирмы Borland. Многочисленные перекрывающиеся друг друга окна, а также сочетание спускающихся и раскрывающихся меню обеспечивают быстрый, интерактивный пользовательский интерфейс. Интерактивная, контекстно-зависимая система подсказки обеспечивает помощь на всех стадиях работы.

После запуска отладчика **td.exe** и загрузки отлаживаемой программы на экране появляется кадр, в котором видны два окна – окно **Module** (модуль) с исходным текстом отлаживаемой программы и окно **Watches** (наблюдения) для отслеживания за ходом изменения заданных переменных в процессе выполнения программы (рис. 3.1). В верхней части кадра представлены **10 кнопок** главного меню отладчика, а в нижней части приводится список функциональных клавиш, позволяющих управлять его работой.

Начальное окно отладчика дает мало информации для отладки программы. Гораздо более информативным является «**окно процессора**», которое вызывается с помощью пункта главного меню **View>CPU** или командой **<Alt>+<V>+<C>** (см. рис 3.2).

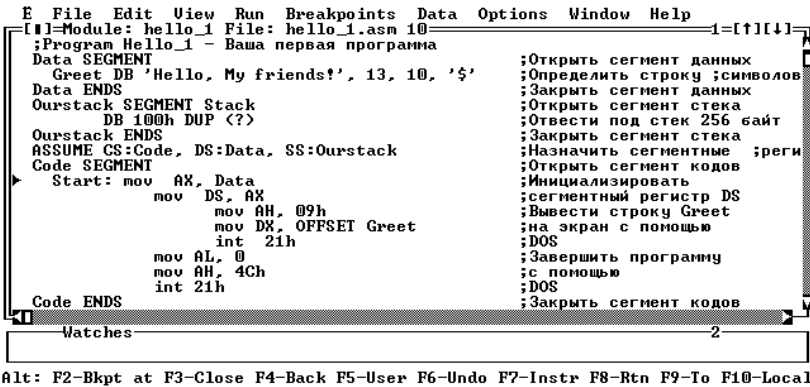


Рис. 3.1 – Начальное окно отладчика с текстом программы

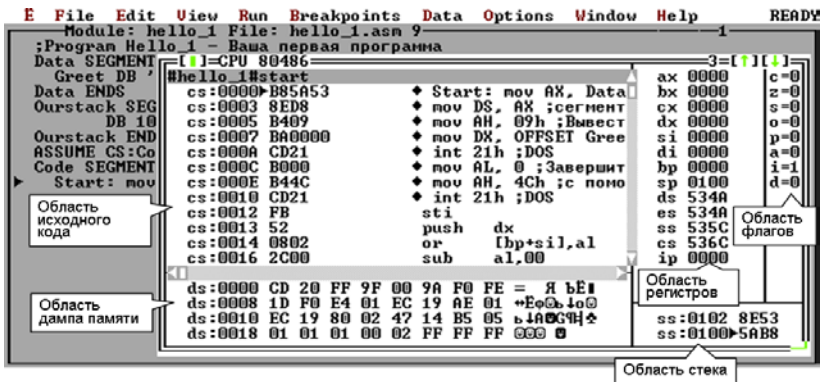


Рис. 3.2 – Окно процессора с внутренними окнами

В окне **CPU (ЦП)** показано состояние центрального процессора. Это окно в свою очередь состоит, из **5 внутренних областей** для наблюдения:

- исходного текста программы на языке ассемблера и в машинных кодах (сегмент кода);
- состояния регистров микропроцессора;
- состояния флагов;
- состояния стека (сегмент стека);
- дампа памяти.

С помощью этого окна можно полностью контролировать ход выполнения отлаживаемой программы. Для того, чтобы можно было работать с конкретной областью окна надо сделать её активной (клавиша **<ТАБ>** или левая кнопка мыши) и перейти в локальное меню выбранной области нажав **<ALT>+<F10>** или правую кнопку мыши.

В области исходного текста или **Code (Код)** для временной коррекции своей программы можно использовать встроенный Ассемблер. Для этого нужно сделать окно активными выбрать пункт локального меню **Assembler**. При этом инструкции вводятся также, как при наборе исходных операторов Ассемблера. Можно также получить доступ к соответствующим данным любой структуры данных, вывода и изменяя их в различных форматах.

В области **регистров** (верхняя область справа от области кода) по умолчанию выводится содержимое 16-и разрядных **регистров центрального процессора**. Если требуется контролировать содержимое 32-х разрядных регистров, то через локальное меню окна следует выбрать опцию **Registers 32-bit Yes**. При необходимости содержимое регистров можно изменять через локальное меню.

Верхней правой областью является **область флагов**, где показано содержимое **восьми флагов центрального процессора**. Значения флагов также можно изменять через локальное меню.

В нижнем правом углу окна **CPU** показано содержимое **стека**. Адрес входа в стек определяется содержимым регистров **SS:SP**.

В области **данных** показано непосредственное содержимое выбранной области памяти. В левой части каждой строки показан **логический адрес данных**, выводимых на данной строке. Адрес выводится в виде пары **SEG:EA**. Значение **SEG** заменяется содержимым регистра **DS**, если значение сегмента совпадает с текущим содержимым регистра **DS**. В правой части каждой строки выводятся символы, соответствующие показанным байтам. Турбо отладчик выводит все печатаемые значения, соответствующие байтовым эквивалентам, поэтому на экране можно увидеть странные символы. Они соответствуют символическому эквиваленту шестнадцатеричных значений байтов, хранящихся в ячейках памяти.

Итак, активизация нужного окна и переход к дополнительному меню позволяют значительно расширить возможности отладчика. Вид этого меню зависит от того, какая область была активна в момент ввода команды. На рис. 3.3 показано дополнительное меню области дампа (отображения) памяти.

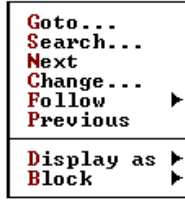


Рис. 3.3 – Дополнительное меню области дампа памяти

Чаще всего используется первый пункт этого меню – **Goto**, с помощью которого можно задать любой адрес, и получить дамп этого участка памяти. На рис. 3.4 изображено содержимое **окна дампа** после ввода начального адреса в виде **DS:0**.

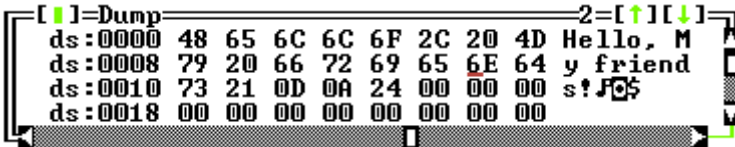


Рис. 3.4 – Окно дампа памяти

3 Подготовка к работе

- 3.1 Изучить методические указания и рекомендованную литературу.
- 3.2 Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

- 4.1 Используя текстовый редактор, создать исходный модуль программы **Prog 3** с помощью шаблона, приведённого ниже. Начальные значения переменных A, B, C, D взять из таблицы 3.1 в соответствии с вариантом. В исходный модуль добавить недостающие комментарии.

Таблица 3.1

Начальные значения переменных для программы Prog 3

№ варианта	Значения переменных				№ варианта	Значения переменных			
	A	B	C	D		A	B	C	D
1	3	9	2Eh	AAh	9	32	6	9h	eh
2	5Ah	2	42	9	10	22h	32	25	10h
3	B5h	55h	15	8	11	32	C1h	6	21
4	22h	7	8	12	12	3Bh	10	12h	9
5	15	1Ah	1Fh	6	13	3Bh	1Fh	11	12
6	3	1Eh	12	22h	14	5	8	10h	0Fh
7	7h	12	1Dh	9	15	12h	12	05h	9
8	5	2Eh	18h	11	16	9	1Ch	8	10h

;Program_3 – Команды передачи данных, вариант 16

Data SEGMENT	;Открыть сегмент данных	
A DB ?	;Зарезервировать место	
B DB ?	;в памяти для	
C DB ?	;переменных	
D DB ?	;A, B, C, D	
Data ENDS	;Закрыть сегмент данных	
Ourstack SEGMENT Stack	;Открыть сегмент стека	
DB 100h DUP (?)	;Отвести под стек 256 байт	
Ourstack ENDS	;Закрыть сегмент стека	
ASSUME CS:Code, DS:Data, SS:Ourstack	;Назначить сегментные регистры	
Code SEGMENT	;Открыть сегмент кодов	
Start: mov AX, Data	;Инициализировать	1
mov DS, AX	;сегментный регистр DS	2
mov A, 9	;Инициализировать	3
mov B, 1Ch	;переменные A, B, C, D	4
mov C, 8	;значениями Вашего	5
mov D, 10h	;варианта	6
mov AL, A		7
mov AH, B		8
xchg AL, AH		9
mov BX, 3E10h		10
mov CX, BX		11
push BX		12
push CX		13
push AX		14
lea SI, C		15
mov AX, SI		16
lea DI, D		17
mov BX, DI		18
pop AX		19
pop CX		20
pop BX		21
mov BX, AX		22
mov A, AL		23
mov B, AH		24
mov C, 0		25
mov AX, 4C00h	;Завершить программу	26
int 21h	;с помощью DOS	27
Code ENDS	;Закрыть сегмент кодов	28
END Start	;Конец исходного модуля.	29

4.2 Создать исполняемый модуль программы **Prog_3.exe** выполнив этапы ассемблирования и компоновки.

4.3 Запустить программу **td.exe** на выполнение. После появления визитной карточки отладчика нажмите клавишу **ENTER**. Обратите внимание на то, что в нижней строке расположена подсказка о назначении функциональных клавиш, в верхней строке перечислены пункты главного меню отладчика. Через меню **View** перейдите в окно **CPU (ЦП)**. Клавишей **ZOOM** измените размер открытого окна.

4.4 Обратите внимание на то, что окно **CPU** разделено рамками на фрагменты (внутренние области), относящиеся к **сегментам кода, данных, стека**, а также к **регистрам и флагам**. В области кода команда по смещению, равному содержимому регистра **IP**, а в области стека данные по смещению, равному содержимому регистра **SP**, отмечены стрелками. Переход из одного внутреннего окна в другое производится клавишей **<TAB>** или мышью.

4.5 Нажатием комбинации клавиш **<Alt>+<F10>** или правой кнопкой мыши, попробуйте открыть окна локальных меню в каждой области окна **CPU**, предварительно сделав их активными. Ознакомьтесь с их содержанием. Закрывайте окна клавишей **<ESC>**.

4.6 Используя клавишу **<F10>**, перейдите в главное меню. Откройте меню **FILE**. Включите режим **OPEN...**. Клавишей **<TAB>** выделите окно **FILES**. Курсорными клавишами выберите имя файла **Prog_3.exe** и загрузите его¹. Сравните информацию, содержащуюся в области кода с листингом вашей программы.

4.7 В окне **CPU** произведите **трассировку программы** (пошаговое выполнение) нажатием клавиши **<F8>**. На каждом шаге контролируйте содержимое регистров, флагов и состояние стека.

4.8 Заново загрузите в отладчик файл **Prog_3.exe**. Включите отображение 32-х разрядных регистров **CPU**. Еще раз произведите **трассировку программы** (пошаговое выполнение) нажатием клавиши **<F8>**, обращая особое внимание на состояния 32-х разрядных регистров.

4.9 Заново загрузите в отладчик файл **Prog_3.exe**. Выполняя программу в пошаговом режиме, заполните таблицу 3.3 для строк программы, указанном в Вашем варианте задания (таблица 3.2).

¹ Приступая к работе с отладчиком, следует убедиться, что в рабочем каталоге имеются и исполняемый (*.EXE), и исходный (*.ASM) файлы.

Таблица 3.2

Варианты заданий			
№ варианта	Строки Prog 3	№ варианта	Строки Prog 3
1	1, 2, 10, 11, 12	9	2, 3, 12, 13, 14
2	2, 3, 9, 10, 11	10	3, 4, 11, 12, 13
3	3, 4, 13, 14, 15	11	4, 5, 14, 15, 16
4	4, 5, 15, 16, 17	12	5, 6, 16, 17, 17
5	5, 6, 17, 18, 19	13	6, 7, 18, 19, 20
6	6, 7, 19, 20, 21	14	7, 8, 20, 21, 22
7	7, 8, 21, 22, 23	15	1, 2, 22, 23, 24
8	1, 2, 23, 24, 24	16	1, 2, 3, 4, 5

Таблица 3.3

Результаты выполнения заданий						
Вариант ...						
Файл и № строки	Команда Ассемблера	Машинный код	Длина машинного кода, байт	Логический адрес в памяти	Физический адрес в памяти	Состояние регистров и флагов
Prog_3 1						AX=..., BX=..., CX=..., DX=..., SP=..., BP=..., SI=..., DI=..., IP=..., DS=..., SS=..., CS=..., ES=...; CF=..., ZF=..., SF=..., OF=..., PF=..., AF=...
Prog_3 2						AX=..., BX=..., CX=..., DX=..., SP=..., BP=..., SI=..., DI=..., IP=..., DS=..., SS=..., CS=..., ES=...; CF=..., ZF=..., SF=..., OF=..., PF=..., AF=...

4.10 Заново загрузите программу **Prog_3** в отладчик. Начните трассировку программы. После инициализации сегментных регистров зафиксируйте их содержимое и составьте образ размещения программы в памяти ЭВМ (см. лабораторную работу №2).

4.11 Определите начальные и конечные адреса сегмента данных, сегмента стека и сегмента кодов. Вычислите длину сегмента данных и сегмента кодов программы в байтах. Проанализируйте файл **Prog_3.map** и сравните результаты Ваших вычислений с цифрами, приведенными в этом файле.

4.12 Просмотрите и зарисуйте область памяти, в которой хранятся данные, объявленные в сегменте данных программы (дамп памяти).

4.13 Прочитайте в трех ячейках памяти начиная с адреса **F000:FFF5h** дату выпуска ПЗУ BIOS в формате месяц/число/год.

4.14 Прочитайте в памяти по адресу **F000:FFFEh** однобайтовый идентификатор модели ЭВМ.

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также фамилии И.О. студента, подготовившего отчёт;

- цель работы;

- листинги программы **Prog_3** с комментариями (см. п.4.1 задания);

- таблицу 3.3 с результатами исследования отладки программы;

- рисунки образов программ **Prog_3** в памяти ЭВМ (см. п.4.9 задания);

- данные файлов *.map и вычисленные начальные и конечные адреса сегментов программы и их длины (см. п.4.10 задания);

- рисунки областей памяти с хранящимися данными (см. п.4.11 задания);

- скрины окна с датой выпуска ПЗУ и идентификатором ЭВМ (см. п. 4.13, 4.14).

6 Контрольные вопросы

6.1 Как записываются общие команды передачи данных на Ассемблере? Что может использоваться в качестве операндов команды?

6.2 Для чего предназначена команда LEA и что является ее операндами?

6.3 Поясните выполнение команд работы со стеком.

6.4 Поясните выполнение команды обмена данными.

6.5 Для чего предназначен отладчик TurboDebugger?

6.6 Объясните смысл пунктов Главного меню в верхней строке отладчика.

6.7 Как загрузить отлаживаемую программу?

- 6.8 Какие окна можно открыть из пункта Главного меню View?
- 6.9 Из каких фрагментов состоит окно CPU?
- 6.10 Что такое локальное меню окна и как его открыть?
- 6.11 Какие функции обеспечивает фрагмент кода (CODE) окна CPU?
- 6.12 Какие функции обеспечивает фрагмент памяти окна CPU?
- 6.13 Какие функции обеспечивает фрагмент регистров окна CPU?
- 6.14 Какие функции обеспечивает фрагмент стека окна CPU?
- 6.15 Какие функции обеспечивает фрагмент флагов окна CPU?
- 6.16 Каким образом можно редактировать ассемблерную программу?
- 6.17 Как осуществляется изменение содержимого оперативной памяти и регистров средствами отладчика?
- 6.18 Как через меню отладчика запустить программу на выполнение?
- 6.19 В каком окне можно наблюдать результат выполнения программы?
- 6.20 Что такое трассировка программы и как она осуществляется в отладчике?

7 Рекомендуемая литература

- 7.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с.135...141.
- 7.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 40...50.
- 7.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К. Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 45...56.

Лабораторная работа №4

Программирование арифметических операций.

Изучение основ работы с TURBO DEBUGGER

1 Цель работы

Программирование задач, выполняющих арифметические вычисления и получение навыков отладки программ средствами отладчика TURBO DEBUGGER.

2 Теоретический материал

2.1 Арифметические команды в Ассемблере

В язык Ассемблера входят пять групп арифметических команд: команды преобразования типов, команды двоичной арифметики, десятичной арифметики, вспомогательные команды и прочие команды с арифметическим принципом действия.

Микропроцессор 80x86 может работать с целыми числами **со знаком** и с целыми **беззнаковыми** числами. Целые беззнаковые числа не имеют знакового разряда, поэтому все его двоичные биты отводятся под мантиссу числа. В числах со знаком под знак отводится самый старший бит двоичного числа: **0 – положительное число; 1 – отрицательное**. Поэтому диапазон значений двоичного числа зависит от его размера и трактовки старшего бита числа. Необходимо помнить, что числа со знаком представляются в **дополнительном коде**.

Таблица 4.1

Диапазон значений двоичных чисел

Размерность поля	Целое число без знака	Целое число со знаком
байт	0...255	-128...+127
слово	0...65 535	-32 768...+32 767
двойное слово	0...4 294 967 295	-2 147 483 648...+2 147 483 647

ADD – целочисленное сложение

Команда **ADD** осуществляет сложение первого и второго операнда, при этом исходное значение первого операнда (**dst**– приёмника) теряется, замещаясь результатом сложения. Второй операнд (**src** – источник) не изменяется.

ADD dst, src; dst: = (dst) + (src).

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме

сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и являться числами со знаком или без знака. Команда воздействует на флаги **OF**, **SF**, **ZF**, **AF**, **PF** и **CF**.

При сложении беззнаковых чисел, когда размерность результата операции выходит за разрядную сетку операндов, могут возникнуть ошибки с определением точного результата. Для индикации переполнения предназначен **флаг переноса CF**, который необходимо проконтролировать при выполнении операции сложения.

При операциях с знаковыми числами нужно учитывать возможный перенос в старший знаковый разряд, так как при этом может измениться знак результата. Для этих целей может помочь анализ **флага переполнения OF**, так как он устанавливается в 1, если происходит перенос в старший знаковый разряд (в 7-ой или 15-ый) для положительных чисел или из старшего знакового разряда для отрицательных чисел.

Совместное использование флагов **CF** и **OF** позволяет контролировать правильность результата при сложении чисел со знаком:

если **CF = OF**, переполнения нет;

если **CF ≠ OF**, есть переполнение и нужно корректировать результат.

SUB – вычитание целых чисел

Команда **SUB** вычитает второй операнд из первого и помещает результат на место первого операнда, т.е.

SUB dst, src; dst = (dst) – (src).

Операнды аналогичны операндам команды целочисленного сложения. Команда воздействует на флаги **OF**, **SF**, **ZF**, **AF**, **PF** и **CF**.

Команды INC и DEC

Команда **INC** (инкремент) прибавляет 1 к операнду (op), в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово:

INC op; op := (op) + 1.

Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги **OF**, **SF**, **ZF**, **AF** и **PF**. Команда не воздействует на флаг **CF**.

Команда **DEC** (декремент) аналогична команде **INC**, за исключением того, что она вычитает единицу из операнда:

DEC op; op:= (op) – 1.

Команды умножения

Для умножения чисел без знака предназначена команда **MUL**, которая имеет следующий формат:

MUL src ;AX:= (AL)*(src) – при умножении байтов,
;DX:AX:= (AX)*src – при умножении слов.

Как видно, второй операнд должен находиться или в регистре-аккумуляторе **AL** (в случае умножения на байт), или в регистре-аккумуляторе **AX** (в случае умножения на слово). После выполнения операции с однобайтовыми числами, 16-и битовый результат записывается в регистр-аккумулятор **AX**; для двухбайтовых чисел произведение длиной в 32 бита формируется в паре регистров **DX:AX** (в **DX** – старшая часть, в **AX** – младшая). Предыдущее содержимое регистра **DX** затирается.

Если содержимое регистра **AX** после однобайтового умножения или содержимое регистра **DX** после двухбайтового умножения не равны 0, флаги **CF** и **OF** устанавливаются в 1. В противном случае оба флага сбрасываются в 0.

В качестве операнда-сомножителя команды **MUL** можно указывать регистр (кроме сегментного) или ячейку памяти. Не допускается умножение на непосредственное значение.

Для умножения чисел со знаком предназначена команда

IMUL src.

Эта команда выполняется так же, как и команда **MUL**. Отличительной особенностью команды **IMUL** является только формирование знака. Если результат мал и умещается в одном регистре (то есть если **CF=OF=0**), то содержимое другого регистра (старшей части) является расширением знака – все его биты равны старшему биту (знаковому разряду) младшей части результата. В противном случае (если **CF=OF=1**) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата.

Команды деления

Команды деления для знаковых и беззнаковых операндов **DIV** и **IDIV** выполняют целочисленное деление, формируя целое частное и целый остаток. Формат команды:

DIV src; $AL := \text{quot}((AX)/(src))$; частное при делении на байт.
 $AH := \text{rem}((AX)/(src))$; остаток при делении на байт.
 $AX := \text{quot}((DX:AX)/(src))$; частное при делении на
 ; слово
 $DX := \text{rem}((DX:AX)/(src))$; остаток при делении на
 ; слово.

При этом делимое должно находиться в регистрах **AX** (в случае деления на байт) или **DX:AX** (в случае деления на слово). **Размер делимого должен быть в два раза больше размеров делителя и остатка.**

После выполнения операции с однобайтовыми делителями, частное записывается в регистр **AL**, остаток – в регистр **AH**; для двухбайтовых делителей – частное в **AX**, остаток в **DX**.

Если делитель равен 0, или если частное не помещается в назначенный регистр, возбуждается прерывание с вектором 0 (деление на 0).

Команды не воздействуют на флаги процессора.

При делении целых чисел со знаком (IDIV), и частное, и остаток рассматриваются как числа со знаком, причём знак остатка равен знаку делимого.

Команды согласования размеров операндов

При выполнении арифметических операций часто используются команды преобразования байта в слово или слова в двойное слово. Эти команды выполняют преобразование над операндом, находящимся в регистре-аккумуляторе:

CBW– преобразовать байт в **AL** в слово в **AX**. При этом старший байт **AX** заполняется значением старшего разряда регистра **AL**:

CBW; $AX := D_7D_7D_7D_7D_7D_7D_7(AL)$
 ; $(AL) := D_7D_6D_5D_4D_3D_2D_1D_0$.

CWD – преобразовать слово в **AX** в двойное слово в **DX:AX**. При этом регистр **DX** заполняется значением старшего разряда регистра **AX**:

CWD; $DX := D_{15}D_{15}D_{15} \dots D_{15}$
 ; $(AX) := D_{15}D_{14}D_{13} \dots D_4D_3D_2D_1D_0$.

2.2 Работа с TURBO DEBUGGER

Если программа скомпонована в режиме /v, то после ее загрузки отладчиком, открывается окно **Module**. Символ <стрелка> показывает на подлежащую исполнению команду. Клавишей **F2** можно расстав-

лять и снимать точки останова (ловушки) (**Breakpoints**) в той строке, где расположен курсор для остановки выполнения программы.

Окно **Inspect** можно открыть из локального меню окна **Module** (**alt-F10**). При этом отладчик запрашивает имя подлежащих контролю переменной или регистра. Контролировать состояния переменных можно также в окнах **Variables** и **Watches**, вызываемых из пункта **View** главного меню.

Окно переменных **Variables** позволяет наблюдать все переменные, доступные в месте останова программы. В локальном окне пункт **Inspect** дает доступ к полной информации о типе, значении и адресе хранения выделенной переменной. Отдельные переменные программист может задать для анализа в окне **Watches**. Для помещения переменной в это окно следует подвести курсор к идентификатору переменной и нажать **Ctrl+W**. Для удаления переменной из окна можно воспользоваться локальным меню, либо клавишей **Delete**.

3 Подготовка к работе

- 3.1 Изучить методические указания и рекомендованную литературу.
- 3.2 Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

4.1 Используя текстовый редактор, создать и отредактировать исходный модуль программы **Prog_4.asm**, которая вычисляет значение **X** в соответствии с вариантом задания. Номер функции и значения переменных **A**, **B** и **C** взять из таблицы 4.2. Значение переменной **D** берётся равным последней цифре номера зачётной книжке. После выполнения операции деления, в дальнейших операциях учитывать только частное.

;Program_4 – Арифметические операции, вариант ...

Data SEGMENT	;Открыть сегмент данных
A DB 1	;Инициализировать
B DB 2	;переменные A, B, C, D, X
C DB 3	
D DB 4	
X DW ?	
Data ENDS	;Закрыть сегмент данных
Ourstack SEGMENT Stack	;Открыть сегмент стека
DB 100h DUP (?)	;Отвести под стек 256 байт
Ourstack ENDS	;Закрыть сегмент стека
ASSUME CS:Code, DS:Data, SS:Ourstack	;Назначить сегментные
	регистры
Code SEGMENT	;Открыть сегмент кодов

```
Start: mov AX, Data      ;Инициализировать
      mov DS, AX        ;сегментный регистр DS
      xor AX, AX        ;Очистить регистр AX
```

*Здесь должны быть команды вычисления
арифметического выражения*

```
      mov AX, 4C00h      ;Завершить программу
      int 21h           ;с помощью DOS
Code ENDS               ;Закрывать сегмент кодов
END Start               ;Конец исходного модуля.
```

Таблица 4.2

Варианты заданий				
№ варианта	Функция	Данные		
		A	B	C
1	$X = \frac{2 * A + B * D}{C - 3}$	64h	14h	-4
2	$X = \frac{D * C}{2 * A + B}$	16h	-50	1Bh
3	$X = \left(1 + \frac{A}{5}\right) * B - C * D$	150	111b	48h
4	$X = \frac{A^2 + D}{C - B}$	15	150h	5
5	$X = (48 + 3 * A) - \frac{B}{C} * D$	5Ah	55h	11h
6	$X = \frac{(B - 25)^2}{A + 1} + (B + D)^2$	-5	31	–
7	$X = (A + B) * (C - 4000) * (D + 212)$	A1h	-150	FB0h
8	$X = (A * B - C * D)^2$	Fh	14	10h
9	$X = \frac{A^2 + B^2}{D - C}$	7	12	-15
10	$X = \frac{(B - C) * A^2}{D - 12}$	5	E2h	225

Продолжение таблицы 4.2

№ варианта	Функция	Данные		
		A	B	C
11	$X = \frac{300 - D + B * C}{A}$	8	26h	-10
12	$X = \frac{65528 - A * B}{(D + C)^2}$	BFh	14h	2
13	$X = \frac{A * (B + 1)}{C} - D$	32	Fh	80
14	$X = 3 * (A - B) + \frac{D}{C}$	99h	D9h	155
15	$X = \frac{(-1) * (D + 1)}{A + B * C}$	Ch	4	9

4.2 Используя компилятор Турбо Ассемблера **tasm.exe** и компоновщик **tlink.exe** с соответствующими ключами, создать файл **prog_4.exe**.

4.3 Загрузите программу в отладчик и в окне **CPU** произведите трассировку программы (пошаговое выполнение) нажатием клавиши **F8**. На каждом шаге контролируйте содержимое регистров и флагов. Запишите вычисленное значение частного и остатка. Убедитесь в правильности вычислений. Результаты пошагового выполнения сведите в таблицу 4.3 и сделайте по ним соответствующие выводы.

4.4 Определите самую длинную и самую короткую команды программы **prog_4.exe**.

4.5 Определите начальные и конечные адреса сегментов кода, данных и стека составленной программы **prog_4.exe**. Вычислите длину сегментов указанной программы в байтах. Составьте и нарисуйте образ программы в памяти ЭВМ.

Таблица 4.3

Результаты выполнения программы

Вариант ...					
№ строки	Команда Ассемблера	Машинный код	Длина машинного кода, байт	Логический адрес в памяти	Состояние регистров и флагов
1					AX=..., BX=..., CX=..., DX=..., SP=..., BP=..., SI=..., DI=..., IP=..., DS=..., SS=..., CS=..., ES=...; CF=..., ZF=..., SF=..., OF=..., PF=..., AF=...
2					AX=..., BX=..., CX=..., DX=..., SP=..., BP=..., SI=..., DI=..., IP=..., DS=..., SS=..., CS=..., ES=...; CF=..., ZF=..., SF=..., OF=..., PF=..., AF=...
.					
.					
.					
Значение переменной X: частное = ... остаток = ...					

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также фамилии И.О. студента, подготовившего отчёт;
- цель работы;
- листинги программы **Prog_4** с комментариями;
- таблицу 4.2 с результатами исследования отладки программы;
- рисунок образа программы **Prog_4** в памяти ЭВМ (см. п.4.4 задания);
- данные файлов *.map и вычисленные начальные и конечные адреса сегментов программы и их длины (см. п.4.4 задания).

6 Контрольные вопросы

- 6.1 Формат команды «сложить», ее операнды.
- 6.2 Формат команды «вычесть», ее операнды.
- 6.3 Формат команды «умножить», ее операнды.
- 6.4 Формат команды «делить», ее операнды.
- 6.5 Каков диапазон беззнаковых чисел допустим в программах 16-ти разрядного микропроцессора?
- 6.6 Каков диапазон чисел со знаком допустим в программах 16-ти разрядного микропроцессора?
- 6.7 Какую информацию содержат арифметические флаги операций?
- 6.8 Какие флаги устанавливаются при выполнении команд «сложить» и «вычесть».
- 6.9 Какие флаги устанавливаются при выполнении команд «умножить» и «делить».
- 6.10 Как выполнить сложение (вычитание) двух операндов, находящихся в памяти?
- 6.11 Как выполнить умножение двух операндов, находящихся в памяти?
- 6.12 Как выполнить деление двух операндов, находящихся в памяти?
- 6.13 С числами какой системы счисления может работать Ассемблер?
- 6.14 Каким образом можно проконтролировать значения переменных при отладке программы?
- 6.15 Как можно контролировать область данных программы, загруженной в память?
- 6.16 Как можно контролировать область стека программы, загруженной в память?
- 6.17 Найдите ошибки в нижеприведенных командах:

MOV AL, E4h; ADD 64, BL; MUL 3Fh; MOV DS, 3F3Fh.

7 Рекомендуемая литература

- 7.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с.165...181.
- 7.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 40...54, 197-280.
- 7.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К. Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 45...60, 320...410.

Лабораторная работа №5

Исследование способов адресации операндов

1 Цель работы

Получение практических навыков использования различных способов адресации операндов. Практическое освоение основных функций TURBO DEBUGGER.

2 Теоретический материал

В выполнении любой команды Ассемблера, за исключением безоперандных, участвуют ее операнды. **Операнды команд могут храниться в регистрах микропроцессора, в ячейках памяти или могут быть указаны непосредственно в команде.** Где бы ни находился операнд его местоположение можно определить с помощью адреса: каждый регистр, ячейку памяти или команду, которой передается управление, можно связать с их адресами. Для поиска операндов команд микропроцессор использует различные способы, которые и **называются способами адресации (режимами адресации).** Информация о том, какие способы адресации будут использоваться, содержится в ассемблерной команде. **Операнды одной и той же команды могут адресоваться по-разному или одинаково.**

Несмотря на то, что в процессоре реализовано более двух десятков различающихся способов формирования адресов операндов, их принято объединять в группы по функциональным признакам, что позволяет провести их систематизацию.

Способы адресации можно разделить на **прямые и косвенные.**

2.1 Прямые способы адресации

Прямая регистровая адресация

Используется в случаях, когда операнд находится в одном из программно-адресуемых регистров микропроцессора.

MOV AX, BX ;переслать содержимое BX в регистр AX.

XOR BL, AL ;сложить по модулю два содержимое BL и AL.

DEC SI ;уменьшить на 1 содержимое регистра SI.

Непосредственная адресация

Операнд, который представляет собой константу размером байт или слово, содержится непосредственно в команде. Такой способ адресации используется для целей загрузки регистров и переменных в памяти, в качестве формирования масок для работы с отдельными бита-

ми, для сравнения операндов с константами и т. д.

MOV CH, 3Eh ;загрузить регистр CH числом 3Eh
MOV AL, 10000000b ;создать в AL маску с 1 в старшем бите
CMP AH, 0FFh ;сравнить содержимое AH с числом FFh

Прямая адресация к памяти

Простейший способ адресации к переменным, находящимся в памяти. Адрес ячейки памяти, в которой находится переменная, указывается в команде (обычно в символической форме) и из нее поступает в код команды, формируя эффективный адрес операнда **EA** без всяких вычислений.

MOV AX, GAMMA ;переслать в AX переменную GAMMA
SUB TEMP, BL ;вычесть из переменной TEMP содержимое BL

2.2 Косвенные способы адресации

Косвенная регистровая адресация (базовая и индексная)

Это адресация к операнду, находящемуся в памяти. При этом эффективный адрес этого операнда **EA**, содержится в одном из **базовых** или **индексных** регистров. В этих случаях, обозначения регистров, содержащих эффективный адрес, заключаются в *квадратные скобки*. Если используются **базовые регистры BP, BX** – то **адресация базовая**, если **индексные регистры SI, DI** – то **адресация индексная**.

При использовании регистров **BX, SI, DI** происходит обращение к операндам, находящимся в **текущем сегменте данных**, сегментный адрес которого обычно хранится в **сегментном регистре DS**, т.е. к операндам с **логическими адресами DS:BX, DS:SI, DS:DI**.

При использовании **регистра BP** происходит обращение к операндам, находящимся в **сегменте стека**, для адресации которого обычно используется **сегментный регистр SS**. Такие операнды имеют **логический адрес** определяемый как **SS:BP**.

ADD AX, [DI] ;сложить содержимое AX и ячейки
;памяти, адресуемой через регистр DI.
MOV [SI], BL ;переслать содержимое BL в ячейку
;памяти по адресу, находящемуся в SI.
CMP byte ptr [BX], 100d ;сравнить содержимое ячейки памяти с
;адресом в BX с числом 100.

Косвенная регистровая адресация со смещением (базовая и индексная со смещением)

Используется для адресации к операндам, находящимся в памяти. Эффективный адрес операнда **EA** определяется как **сумма содержимого одного из базовых или индексных регистров BX, BP, SI, DI и константы, указанной в команде**, которая называется **смещением** (displacement). Смещение может быть числом или адресом. При использовании регистров **BX, SI, DI** обращение происходит **в текущий сегмент данных**, а при использовании **BP** – **в текущий сегмент стека**.

Такую адресацию удобно использовать для обращения к элементам структур данных, когда смещение известно заранее, а базовый (начальный) адрес структуры вычисляется при выполнении программы. Например, если в сегменте данных объявлен массив из 20 символов **@**

```
ARRAY DB 20 DUP ('@'),
```

то обращение к элементам этого массива можно выполнять следующим образом:

```
LEA SI, ARRAY ;загрузить в SI начальный адрес массива
                ;ARRAY.
MOV AL, [SI + 9] ;переслать 9-й элемент массива в регистр AL.
ADD [SI] 5, 0Fh ;сложить 5-й элемент массива с числом Fh.
MOV 8 [SI], AH ;переслать содержимое AH в 8-й элемент
                ;массива.
```

При работе с блоками данных (массивами) необходимо помнить, что индексация элементов начинается с нуля. Это значит, что начальный элемент массива будет иметь нулевой индекс.

Из примера видно, что смещение может быть задано разными способами, независимо от этого происходит его **сложение с содержимым, указанного в команде регистра SI**. В этом примере можно обойтись и без команды **LEA** (загрузить значение **EA**), если предварительно занести в **SI** индекс начального элемента массива и несколько изменить следующие за ней команды:

```
MOV SI, 0 ;загрузить в SI индекс начального
           ;элемента массива ARRAY
MOVAL, ARRAY [SI+9] ;переслать девятый элемент массива в
                    ;регистр AL.
ADD ARRAY [SI] 5, 0Fh ;сложить пятый элемент массива с числом Fh.
MOV ARRAY 8 [SI], AH ;переслать содержимое AH в восьмой
                    ;элемент массива.
```

Применение базовой адресации со смещением при работе со стеком можно проиллюстрировать примером передачи параметров подпрограмме через стек:

```

;Основная программа
PUSH DS           ;сохранить в стеке содержимое трёх
PUSH ES           ;регистров DS, ES, SI, через которые
PUSH SI           ;передаются параметры подпрограмме.
CALL ROUTE        ;вызов подпрограммы.
;Подпрограмма ROUTE
MOV BP, SP        ;загрузить в BP адрес входа в стек.
MOV AX, 2 [BP]    ;извлечь из стека содержимое SI.
MOV BX, 4 [BP]    ;извлечь из стека содержимое ES.
MOV CX, 6 [BP]    ;извлечь из стека содержимое DS.
    
```

В отличие от использования команд **POP**, в данном случае не происходит удаление данных из стека при извлечении. Для этого в регистр **BP** копируется адрес входа в стек и используются команды **MOV** с базовой адресацией со смещением.

Базово-индексная адресация

Используется для нахождения операндов в памяти, причем **эффективный адрес ЕА вычисляется как сумма содержимого двух регистров (базового и индексного), указанных в команде.**

При этом могут использоваться следующие пары регистров:

```

[BX] [SI] – адрес вычисляется как DS: [BX] [SI];
[BX] [DI] – адрес вычисляется как DS: [BX] [DI];
[BP] [SI] – адрес вычисляется как SS: [BP] [SI];
[BP] [DI] – адрес вычисляется как SS: [BP] [DI].
    
```

Удобно использовать при работе с массивами: в одном из базовых регистров находится начальный адрес массива, а в другом индекс элемента, к которому необходимо обратиться.

Базово-индексная адресация со смещением

Используется для нахождения операндов в памяти, причем **эффективный адрес ЕА вычисляется как сумма содержимого двух регистров (базового и индексного) и смещения, указанного в команде.** Этот режим адресации является наиболее гибким, так как два компонента адреса позволяют реализовать наиболее широкие возможности по адресации операндов, например, обращение к элементам двумерного массива (матрице).

Пусть в сегменте данных создан массив из 20 символов (по 10 в строке):

```
MAS DB 'QWERTYUIOP'
      DB 'ЙЦУКЕНГШЦЗ'
```

Для обращения к 5-му элементу из второй строки массива (символ **Е**), нужно написать следующую последовательность команд:

```
MOV BX, 10           ;загрузить в BX число байт в строке.
MOV SI, 4            ;загрузить в SI индекс элемента второй
                    ;строки.
MOV AL, MAS [BX] [SI] ;переслать 5-й элемент второй строки в AL
```

3 Подготовка к работе

- 3.1. Изучить методические указания и рекомендованную литературу.
- 3.2. Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

4.1 Составить программу **Prog_5** для выполнения указанных в таблице 5.1 операций, выбранных в соответствии с вариантом задания. В качестве сегмента данных для программы **Prog_5** используйте приведенный ниже шаблон и упрощенный способ определения сегментов.

; шаблон сегмента данных для Prog_5

```
.Data
var_1 DB 11000110b      ;начало сегмента данных
                        ;определить переменную var_1
                        ;размером байт с начальным
                        ;значением 11000110b
var_2 DW 9FFEH          ;определить переменную var_2
                        ;размером слово с начальным
                        ;значением 9FFEH
var_3 DB ?              ;определить переменную var_3
                        ;размером байт не задавая ее
                        ;начального значения
var_4 DW ?              ;определить переменную var_3
                        ;размером слово не задавая ее
                        ;начального значения
N_1 DD 0FF00FFEEh       ;определить переменную N_1
                        ;размером двойное слово с
                        ; начальным значением
                        ;FF00FFEEh
N_2 DD ?                ;определить переменную N_2
                        ;размером двойное слово не
                        ;задавая ее начального значения
```

String DB 'Assembler', '\$'	;определить строку символов ;String, каждый символ которой ;имеет размер байт с начальным ;значением Assembler
M1 DB 7, 9, 28, 46, 39, 31, 20, 25	;определить массив с именем M1 ;состоящий из восьми числовых ;элементов размером байт
M2 DB 12, 15, 7, 25, 31, 38, 20, 63	;определить массив с именем M1 ;состоящий из восьми числовых ;элементов размером байт
Z1 DW 48, 256, 300, 511, 31, 512	;определить массив с именем Z1 ;состоящий из шести числовых ;элементов размером слово
Z2 DW 0EEh, 99Fh, 300h, 51AAh	;определить массив с именем Z2 ;состоящий из четырех числовых ;элементов размером слово
SIM DB 'QWERTYUIOP' DB 'ЙЦУКЕНГШЦЗ' DB 'POIUYTREWQ'	;создать массив с именем SIM из ;трех строк по 10 символов в ;каждой

- 4.2 Отладить составленную программу.
- 4.3 Определить способы адресации для всех операндов команд.
- 4.4 Определить самую короткую и самую длинную команды в байтах, их логические и физические адреса.

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также фамилии И.О. студента, подготовившего отчёт;
- цель работы;
- вариант задания;
- листинги программы **Prog 5** с комментариями. В комментариях следует указать используемые во всех командах способы адресации;
- размеры в байтах самой короткой и самой длинной команды, а также их логические и физические адреса.

Таблица 5.1

Варианты заданий			
№ варианта	Операция	1-ый операнд или получатель	2-ой операнд или источник
1	пересылка пересылка OR сложение	регистр переменная в памяти var_3 переменная в памяти var_2 1-й элемент Z1	Адрес строковой переменной 3-й символ в строке символов регистр 3-й элемент Z2
2	пересылка AND сложение пересылка	регистр 6-й элемент Z1 переменная в памяти var_2 индексный регистр	2-й элемент Z2 2-й элемент Z1 переменная в памяти var_1 адрес массива SIM
3	пересылка AND вычитание пересылка	базовый регистр переменная в памяти var_2 2-й элемент блока данных Z2 расширенный регистр	адрес блока данных Z2 3-й элемент блока данных Z2 константа 0AA11h переменная в памяти N_1
4	пересылка пересылка сложение OR	индексный регистр регистр переменная в памяти var_2 переменная в памяти var_1	адрес блока данных SIM 3-й элемент 2-й строки SIM 3-й элемент блока данных Z2 константа 3Fh
5	пересылка пересылка пересылка пересылка	расширенный регистр переменная в памяти N_2 в стек в стек	переменная N_1 расширенный регистр 2-й элемент массива Z1 4-й элемент массива Z1
6	пересылка OR пересылка сложение	переменная в памяти var_4 переменная в памяти var_4 1-й элемент блока данных Z2 3-й элемент блока данных Z2	константа 83AAh 3-й элемент блока данных Z1 переменная в памяти var_4 1-й элемент блока данных Z2
7	пересылка пересылка вычитание AND	базовый регистр переменная в памяти var_4 3-й элемент блока данных Z1 переменная в памяти var_2	адрес блока данных Z2 3-й элемент блока данных Z2 4-й элемент блока данных Z2 константа 8000h
8	пересылка пересылка сложение пересылка	1-й элемент блока данных M1 3-й элемент блока данных Z2 2-й элемент блока данных Z2 в стек	5-й элемент блока данных M1 переменная в памяти var_2 регистр 2-й элемент блока данных Z2
9	пересылка пересылка AND сложение	регистр 3-й элемент блока данных M2 регистр 16 бит регистр 16 бит	адрес блока данных SIM 3-й элемент 3-й строки SIM константа 10A0h переменная в памяти var_2
10	сложение пересылка пересылка пересылка	переменная в памяти var_1 4-й элемент блока данных M1 в стек в стек	2-й элемент блока данных M2 переменная в памяти var_1 1-й элемент блока данных Z2 4-й элемент блока данных Z2
11	пересылка пересылка пересылка сложение	сегментный регистр ES 1-й элемент блока данных Z2 3-й элемент 3-й строки SIM переменная в памяти var_1	константа 0FF5Fh переменная в памяти var_2 3-й элемент 1-й строки SIM 3-й элемент 1-й строки SIM

Продолжение таблицы 5.1

№ варианта	Операция	1-ый операнд или получатель	2-ой операнд или источник
12	пересылка сложить пересылка сложение	1-й элемент блока данных M2 4-й элемент блока данных M2 расширенный регистр расширенный регистр	константа байт 0F8h 1-й элемент блока данных M2 переменная в памяти N_1 переменная в памяти N_1
13	пересылка пересылка сложение AND	3-й элемент блока данных Z1 регистр 1-й элемент блока данных Z1 переменная в памяти var_2	константа слово 0h 5-й элемент блока данных Z1 2-й элемент блока данных Z2 1-й элемент массива Z2
14	пересылка сложение пересылка обмен	расширенный регистр переменная N_1 3-й элемент массива M2 2-й элемент массива M1	переменная N_1 расширенный регистр 1-й элемент массива M1 3-й элемент массива M1
15	пересылка OR вычитание пересылка	переменная в памяти var_4 переменная в памяти var_1 3-й элемент массива Z1 в стек	1-й элемент блока данных Z1 константа байт 0FFh 2-й элемент массива Z1 3-й элемент массива Z1

6 Контрольные вопросы

- 6.1 В чем заключается принцип сегментации памяти?
- 6.2 Что означает адрес байта, слова и двойного слова?
- 6.3 Каков минимальный объем адресуемого участка памяти?
- 6.4 Что такое регистровая адресация операндов?
- 6.5 Что используется для прямой адресации ячеек памяти?
- 6.6 Где располагаются операнды при непосредственной адресации?
- 6.7 Что используется для косвенной адресации ячеек памяти?
- 6.8 Какие регистры используются и как вычисляется адрес при базовой адресации?
- 6.9 Какие регистры используются и как вычисляется адрес при индексной адресации?
- 6.10 Какие регистры используются и как вычисляется адрес при базовой адресации со смещением?
- 6.11 Какие регистры используются и как вычисляется адрес при индексной адресации со смещением?
- 6.12 Как вычисляется адрес ячейки памяти при базово-индексной адресации?
- 6.13 Как вычисляется адрес ячейки памяти при базово-индексной адресации со смещением?
- 6.14 Какой способ адресации операндов обеспечивает самый короткий формат команды?
- 6.15 Какой способ адресации является самым сложным?

6.16 Каким образом процессоры i80x86 обращаются к портам ввода/вывода?

7 Рекомендуемая литература

7.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с. 94...97.

7.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 50...73.

7.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К. Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 56...82.

Лабораторная работа №6

Работа с подпрограммами и процедурами

1 Цель работы

Практическое овладение навыками составления и отладки подпрограмм и процедур на языке Ассемблера.

2 Теоретический материал

2.1 Подпрограммы и процедуры

Подпрограммы и процедуры являются одним из средств разработки модульных программ. Они представляет собой законченную командную последовательность к которой можно обращаться из любого места программы с помощью команд вызова **CALL**.

Достоинства использования подпрограмм-процедур объясняется следующим:

- сложную программу можно разбить на сравнительно небольшие и достаточно простые модули, разработка и отладка которых может производиться автономно несколькими программистами;
- в виде процедур оформляются фрагменты программ, которые используются многократно, следовательно использование процедур сокращает длину программ;
- из отлаженных процедур формируются библиотеки, которые можно использовать в других программах.

Организируются **подпрограммы** очень просто:

```
метка:                                ;начало подпрограммы
.....
      команды, входящие
      в подпрограмму
      (тело подпрограммы)
.....
RET                                  ;возврат из подпрограммы.
```

- первая команда подпрограммы помечается **меткой**, которая служит точкой входа в подпрограмму;
- завершается подпрограмма командой возврата **RET**.
- вызов подпрограммы производится командой вызова **CALL**, которая имеет следующий формат:

CALL <метка>.

Любую **подпрограмму** можно оформить в виде **процедуры**. При этом следует использовать следующие правила:

- Начинается процедура директивой **PROC** (Procedure), а завершается директивой **ENDP** (END Procedure), между которыми располагается тело процедуры.

- Директивы начала и окончания процедуры обязательно должны иметь **имя**, которое является именем процедуры. Имя используется как метка первой команды процедуры и означает точку входа в процедуру.

- Некоторые процедуры могут требовать передачу из вызывающей программы входных параметров и возвращать в нее результаты вычислений. В этом случае их нужно оформлять в виде списка входных и возвращаемых параметров (см. п. 2.3).

```
имя PROC [тип]                ;начало процедуры
      [ARG список аргументов] ;входные параметры
      [RETURN список элементов] ;возвращаемые параметры
```

```
.....
```

тело

процедуры

```
.....
```

```
имя ENDP                ;конец процедуры.
```

- После ключевого слова **PROC** указывается **тип процедуры** **NEAR** (ближняя) или **FAR** (дальняя) (по умолчанию принимается тип **NEAR**). Если процедура находится в том же сегменте кода, что и вызывающая команда, она относится к типу **NEAR**. Если процедура и вызывающая команда находится в разных сегментах, то она относится к типу **FAR**.

- Для вызова процедуры используется команда вызова **CALL**, которая имеет следующий формат:

CALL <имя процедуры>.

Если вызываемая процедура относится к типу **NEAR**, то генерируется короткая команда внутрисегментного вызова, т. е. машинный код команды **CALL** содержит внутрисегментное смещение точки входа в процедуру, а **адрес возврата** (содержимое указателя команд **IP**) **автоматически помещается в стек**.

Если вызываемая процедура относится к типу **FAR**, то генерируется длинная команда межсегментного вызова, т. е. машинный код команды **CALL** содержит сегментный адрес и внутрисегментное сме-

шение точки входа в процедуру, а **адрес возврата** (содержимое регистра **CS** и указателя команд **IP**) **автоматически помещается в стек**.

– Тело процедуры должно завершаться безоперандной командой возврата **RET**. Эта команда восстанавливает в регистрах микропроцессора запомненный в стеке **адрес возврата**. В процедурах типа **NEAR** восстанавливается содержимое указателя команд – регистра **IP**, а в процедурах типа **FAR** содержимое двух регистров **CS** и **IP**. Все сказанное для команд вызова **CALL** и возврата **RET** справедливо и для организации подпрограмм.

Элементарный пример процедуры, которая заносит в регистр **CX** среднее значение содержимого регистров **AX** и **DX** приведен ниже:

Average PROC NEAR	;начать процедуру с именем Average
MOV CX, AX	;переслать (AX) в (CX)
ADD CX, DX	;сложить (CX) и (DX)
RCR CX, 1	;сдвинуть (CX) вправо на один разряд
RET	;возврат из процедуры
Average ENDP	;закрывать процедуру именем Average

Вызов этой процедуры производится из любого места программы командой **CALL Average**.

Поскольку и в вызывающей программе, и в процедурах используются одни и те же регистры микропроцессора, важно, чтобы при возврате в вызывающую программу они оказались немодифицированными. Это решается выполнением двух действий:

– сохранение содержимого необходимых регистров в стек в самом начале процедуры. Для этого используются команды **PUSH** или команда **PUSHA** (в случае использования расширенной системы команд);

– извлечения из стека содержимого всех сохраненных регистров с помощью команд **POP** или **POPA** (в случае использования расширенной системы команд).

2.2 Размещение процедур в программе

Размещаться процедуры могут в любом месте программы, однако программист должен принимать соответствующие меры, защищая ее от несанкционированного выполнения. Самым простым способом такой защиты являются расположение процедур выше тех программ, которые их вызывают, а также отсутствие вложенных процедур (когда одна процедура полностью находится внутри другой).

В виде одной процедуры можно оформить сегмент кодов не-сложной программы, например так, как в приводимом ниже примере.

При этом команда возврата **RET** – отсутствует, а имя этой процедуры должно быть указано в директиве **END**, завершающей исходный модуль программы.

TITLE Prog_1	;Пример регистровых операций.
.MODEL SMALL	;Модель памяти ближнего типа.
.STACK 100h	;Отвести под стек 256 байт.
.CODE	;Открыть сегмент кодов.
Begin PROC NEAR	;Начать процедуру с именем Begin.
PUSH DS	;Сохранить DS в стеке.
SUB AX, AX	;Очистить AX.
PUSH AX	;Сохранить AX в стеке.
MOV AX, 0123h	;Передать константу в регистр.
ADD AX, 0025h	;Сложить (AX) с константой.
MOV BX, AX	;Передать из регистра в регистр.
ADD BX, AX	;Сложить содержимое регистров AX и BX.
MOV CX, BX	;Передать из регистра в регистр
SUB CX, AX	;Вычесть (AX) из (CX).
NOP	;Нет операции (задержка).
MOV AX, 4C00h	;Выход
INT 21h	;в DOS.
Begin ENDP	;Закрывать процедуру.
END Begin	;Закрывать программу.

2.3 Входные и выходные параметры процедур

Процедура обычно разрабатывается как функциональный блок, который формирует набор выходных данных путем преобразования определенного набора входных данных. Значения, которые процедура использует как входные данные, называются **параметрами**. Параметрами могут быть численные значения, адреса, содержимое ячеек памяти и т. д. Имеется несколько способов передачи параметров процедурам. Один из простых способов заключается в том, чтобы значения параметров разместить в регистрах микропроцессора, например так, как это сделано в процедуре **Average**. Два входных параметра передаются процедуре через регистры **AX** и **DX**. Эти параметры должны быть загружены в указанные регистры до вызова процедуры.

Еще один способ передачи параметров связан с использованием общей области данных, доступной и вызывающей программе и процедуре. В большинстве программ на языках высокого уровня передача параметров процедурам организуется через стек. В ассемблерных программах такой способ организуется также сравнительно просто. Выбор способа передачи параметров процедуре зависит от конкретных функций выполняемых ею.

Процедура, которая возвращает одно значение, называется **процедурой-функцией**. Примером такой процедуры является рассмотренная **Average**, которая возвращает среднее значение в регистр **CX**. Чтобы упростить объединение исходных модулей, составленных на языках высокого уровня и ассемблере, желательно использовать следующее соглашение:

- значение переменной типа **WORD** возвращается в аккумуляторе **AX**;
- значение переменной типа **BYTE** возвращается в аккумуляторе **AL**;
- короткий указатель адреса (смещение) возвращается в регистре **BX**;
- длинный указатель адреса (база : смещение) возвращается в паре регистров **ES : BX**.

2.4 Вычисление полиномов

Для вычисления полиномов n -й степени вида

$$Y = a_1 X^n + a_2 X^{n-1} + \dots + a_n X + a_{n+1}$$

удобно использовать формулу Горнера

$$Y = \left(\dots \left((a_1 X + a_2) X + a_3 \right) X + \dots + a_n \right) X + a_{n+1}.$$

Если выражение, стоящее внутри скобок, обозначить через Y_i , тогда значение выражения в следующих скобках Y_{i+1} можно вычислить, используя рекуррентную формулу $Y_{i+1} = Y_i X + a_{i+1}$. Значение полинома Y получается после повторения этого процесса в цикле n раз. Начальное значение Y_1 должно быть равно a_1 , а цикл следует начинать с $i=2$.

Приведенный на рисунке 6.1 алгоритм, позволяет вычислить значение полинома n -й степени, значение которого находится в диапазоне $-32768 \dots +32767$. Вычисления сопровождаются выводом на экран сообщений, характеризующих результат:

- если $Y \geq 0$ – “**Result plus**”;
- если $Y < 0$ – “**Result minus**”;
- если $-32768 > Y > 32767$ – “**Overflow**”.

Исходными данными являются:

- коэффициенты полинома a_1, a_2, \dots, a_5 , которые хранятся в памяти в виде одномерного массива **MAS_A**. Размер каждого элемента массива – **WORD**;
- N – переменная для хранения порядка полинома;
- X – переменная для хранения аргумента X полинома;
- **MES_1** – строка символов ‘**Overflow**’;

- **MES_2** – строка символов '**Result minus**';
- **MES_3** – строка символов '**Result plus**';
- **Y** – двухбайтовая переменная для хранения результата вычисления полинома.

Регистры 16-ти разрядного микропроцессора распределены следующим образом:

- **регистр SI** используется для хранения индексов элементов массива **MAS_A**;
- **регистр CX** используется как счетчик циклов, который нужно повторить **N** раз;
- **регистры DX** и **AX** используются для формирования результата вычисления полинома: в **DX** формируется старшее слово результата (оно не используется), в **AX** – младшее слово результата.

В соответствии с **формулой Горнера**, основными операциями при вычислении полинома являются умножение и сложение, которые выполняются в цикле. Вычисление произведения $a_i X$ производится по команде **IMUL X** (блок 3). Результат умножения – двоичное число со знаком помещается в пару регистров: в **DX** – старшая часть, в **AX** – младшая часть. Если в старшей половине (в регистре **DX**) находятся значащие цифры, значит результат выходит за диапазон $-32768...+32767$, а флаги **CF** и **OF** устанавливаются в 1. **Эта ситуация рассматривается как переполнение.** Проверка на переполнение производится в блоке 4, и если переполнение есть (**OF** = 1), то на экран выводится сообщение '**Overflow**' (блок 14) и работа программы завершается.

Если переполнения нет, то к полученному в регистре **AX** произведению прибавляется значение a_{i+1} (блок 6). Для этого содержимое регистра указателя индекса **SI** требуется увеличить на 2 (массив **MAS_A** объявлен как **WORD**) (блок 5). В блоке 7, получившаяся сумма вновь проверяется на переполнение с помощью флага **OF**. Если **OF** = 1 (переполнение), то на экран выводится сообщение '**Overflow**' и работа программы завершается. Если переполнения нет, то вычисления продолжают. В блоках 8 и 9 содержимое **счетчика циклов CX** уменьшается на 1 и если содержимое **CX** не равно 0, то происходит переход к началу цикла. После четырех проходов тела цикла содержимое счетчика **CX** = 0, происходит выход из цикла и результат, сформированный в регистре **AX**, пересылается в переменную **Y** (блок 10).

В блоке 11 проверяется знак результата и если он отрицательный, то на экран выводится сообщение **MES_2 'Result minus'** (блок 12). В противном случае выводится **MES_3 'Result plus'** (блок 15). В блоке 13 осуществляется стандартный выход в DOS.

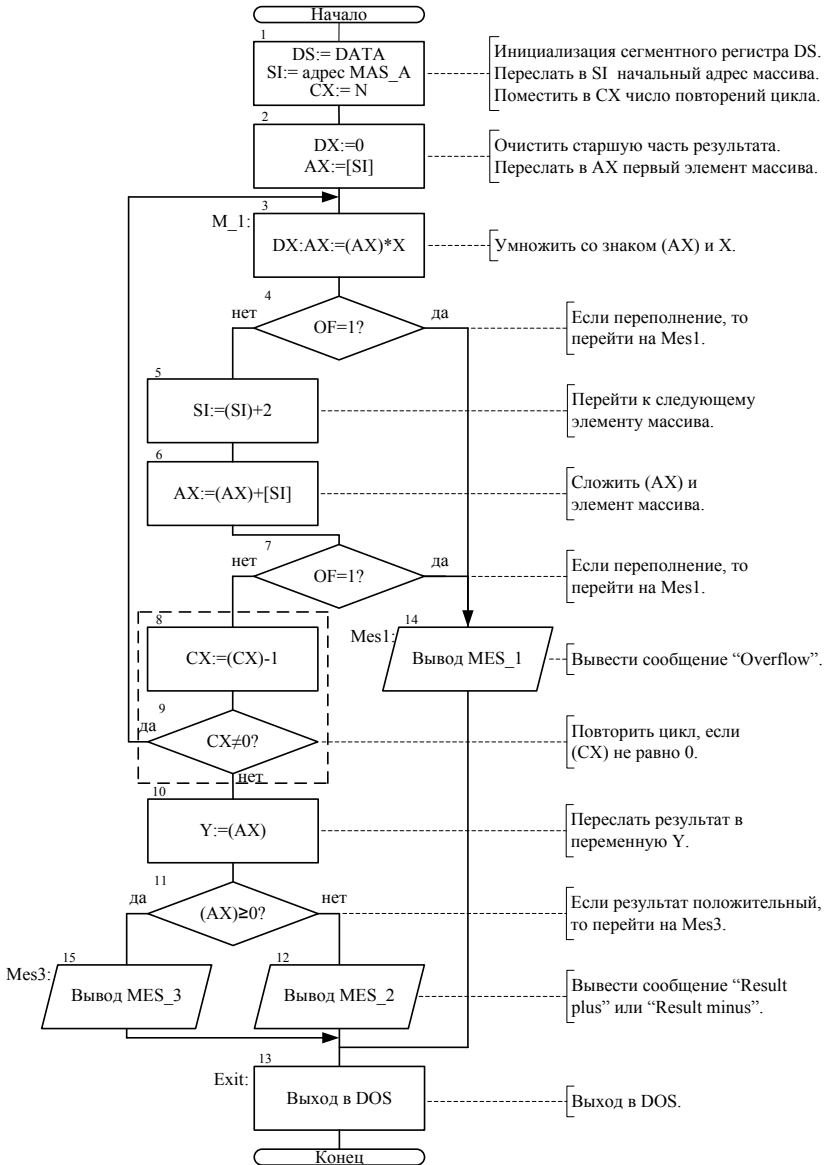


Рис.6.1 – Алгоритм вычисления полинома

2.5 Преобразование двоичных чисел в ASCII коды

Десятичные числа, которые используются в программах, хранятся в памяти или в регистрах микропроцессора либо в **двоичном виде** (двоичнокодированные беззнаковые и знаковые числа), либо в виде **двоично-десятичных чисел** (BCD – Binary-Coded Decimal) упакованного или неупакованного формата. Однако для того, чтобы вывести десятичное число на экран дисплея, необходимо каждую его цифру представить в виде символа. Для кодировки символов клавиатуры (в том числе и цифр) в Ассемблере используются **коды ASCII**, значения которых приведены в таблице 7.3 лабораторной работы №7.

Процесс преобразования **двоичнокодированных чисел в коды ASCII** заключается в последовательном целочисленном делении двоичного числа на 10d и выделении остатков деления до тех пор, пока частное от деления не окажется равным 0. Все получившиеся остатки образуют двоичные коды десятичных цифр, начиная с младшего разряда. Затем эти коды преобразуются в формат ASCII вписыванием числа **3d = 0110b** в старшую тетраду каждого байта.

Алгоритм, реализующий описанный процесс, приведен на рисунках 6.2 и 6.3. При этом следует учитывать:

- исходное преобразуемое число храниться в переменной **Number**;
- информация о знаке числа храниться в переменной **Sign**;
- преобразованное число в символьном виде храниться в переменной **Y_ASCII**;
- регистр **CX** – счетчик циклов;
- регистр **SI** – указатель адреса символов в переменной **Y_ASCII**.

Функционально алгоритм можно разделить на три участка. В блоках 1...5 выполняются подготовительные операции и определяется знак исходного числа, которая заносится в переменную **Sign**. Для этого исходное число **Number** помещается в регистр **AX** (бл.2) и по умолчанию считается положительным, т.е. в переменную **Sign** заносится **символ пробела** (знак плюс опускаем) (бл.3). Проверка знака осуществляется по флагу **SF**, который устанавливается после выполнения сравнения (**AX**) с 0 (бл.4). Если **SF = 0**, то число считается положительным и начинается его преобразование. В противном случае в переменную **Sign** записывается **символ минус** ('-'), и исходное число преобразуется из дополнительного в прямой двоичный код (бл.6).

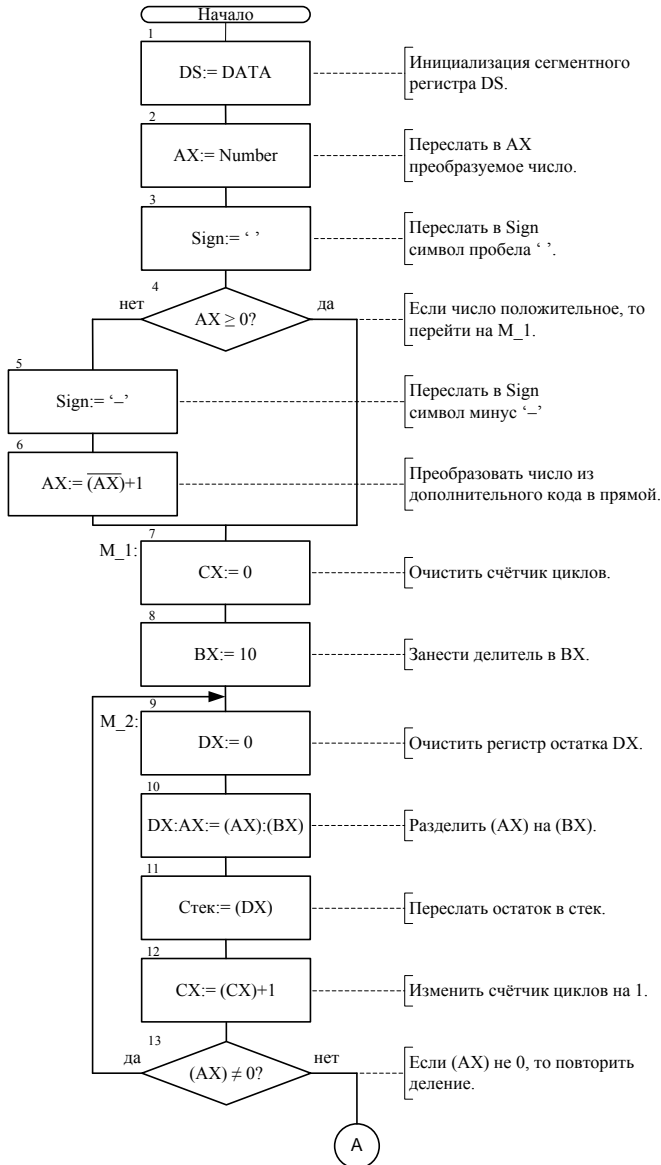


Рис. 6.2 – Алгоритм преобразования двоичного числа в коды ASCII

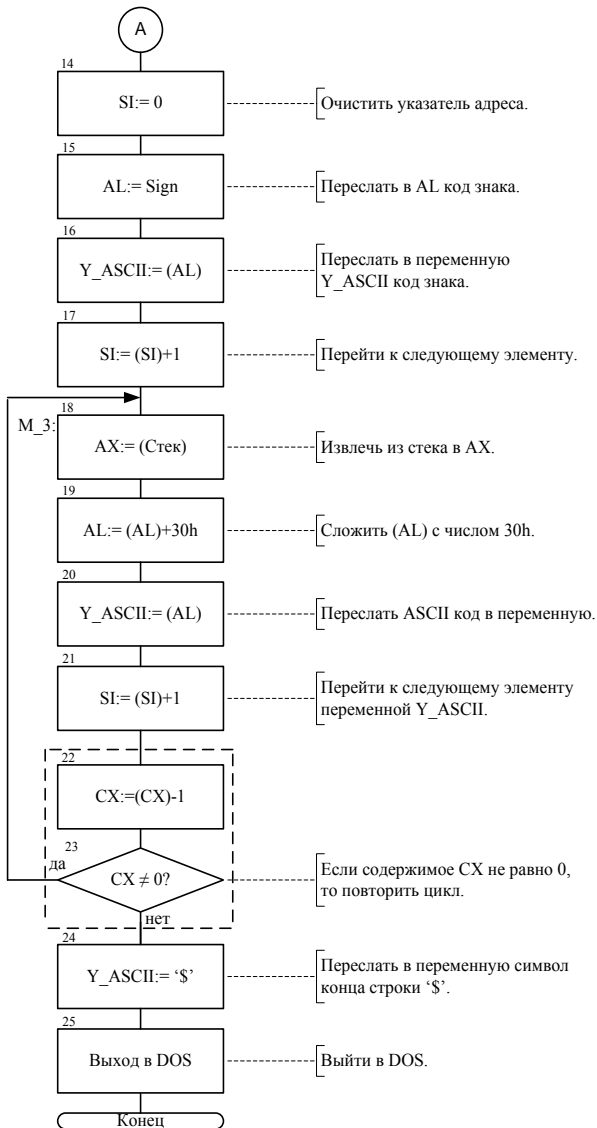


Рис. 6.3 – Продолжение алгоритма преобразования двоичного числа в коды ASCII

В блоках с 7 по 13 выполняется последовательное деление исходного двоичнокодированного числа на **10d** и сохранение остатков деления в **стеке**. Как известно, деление выполняется по команде **DIV src**, где **src** является делителем. Делитель помещается в регистр **BX** (бл.8). **Поскольку размер делимого должен быть в два раза больше размера делителя (регистра BX), оно должно размещаться в паре регистров DX:AX.** Делимым является исходная переменная **Number**, которая имеет размер **WORD** и размещается в аккумуляторе **AX** (бл.2 или бл.6). Это значит, что старшее слово делимого следует установить нулевым, т.е. регистр **DX** – очистить (бл.9). После выполнения деления в бл.10 в **регистре DX образуется целочисленный остаток, а в регистре AX – целое частное.** Остаток сохраняется в стеке (бл.11). В регистре **CX**, который является счетчиком циклов, подсчитывается число последовательных делений (бл.12). Деление продолжается до тех пор, пока частное не станет равным 0 (бл.13).

На третьем этапе алгоритма формируется переменная **Y_ASCII**, которая содержит символы знака и цифр числа. При этом регистр **SI** используется в качестве указателя адреса элементов символьной переменной. В блоках 14...17 в переменную **Y_ASCII** пересылается знак числа. **Сохраненные в стеке остатки деления** (двоичные коды цифр числа) **извлекаются из стека** (бл.18), **преобразуются в ASCII коды** (бл.19) **и пересылаются в символьную переменную Y_ASCII** (бл.20). Эти операции выполняются в цикле до тех пор, пока содержимое счетчика циклов **CX** не станет равным 0 (бл.22, 23). После выхода из цикла, преобразование завершается **пересылкой в переменную Y_ASCII символа конца строки '\$'** (бл.24). Программа завершается выходом в DOS (бл.25).

3 Подготовка к работе

- 3.1. Изучить методические указания и рекомендованную литературу.
- 3.2. Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

- 4.1 Используя, описанный в разделе 2, алгоритм вычисления полинома разобрать исходный текст программы **POLINOM** (см. ниже). Создать и отладить исполняемый модуль программы **POLINOM**, выполнив этапы ассемблирования и компоновки. Добавить в исходный модуль программы недостающие комментарии.
- 4.2 Упростить программу, реализовав трижды используемый вывод сообщений на экран в виде **подпрограмм**.

4.3 Отредактировать исходный модуль программы **POLINOM** для своего варианта задания (таблица 6.1). Создать и отладить исполняемый модуль программы **POLY_X** (X – номер варианта), выполнив этапы ассемблирования и компоновки. В случае появления переполнения выяснить причину и устранить ее уменьшив коэффициент a_1 .

TITLE POLINOM

;Программа вычисления полинома вида $Y=a_1X^n + a_2X^{n-1} + \dots + a_nX + a^{n+1}$

;Входные параметры:

; коэффициенты полинома a_i в массиве **MAS_A**

; порядок полинома **N**

; аргумент полинома **X**

; Выходные параметры:

; вычисленное значение полинома **Y**

; сообщения **MES_1**, **MES_2**, **MES_3**

.MODEL SMALL

;Модель памяти ближнего типа.

.STACK 100h

;Отвести под стек 256 байт.

.DATA

;Открыть сегмент данных.

Mas_A DW -3, 3, -6, 9, -20

;Коэффициенты полинома.

N DW 4

;Порядок полинома равен 4.

X DW 10

;Аргумент полинома равен 10.

Y DW (?)

;Результат вычисления полинома.

Mes_1 DB 'OVERFLOW', 13, 10, '\$'

;Сообщение MES_1.

Mes_2 DB 'RESULT MINUS', 13, 10, '\$'

;Сообщение MES_2.

Mes_3 DB 'RESULT PLUS', 13, 10, '\$'

;Сообщение MES_3.

;

.CODE

;Открыть сегмент кодов.

Start: mov AX, @Data

mov DS, AX

lea SI, Mas_A

mov CX, N

xor DX, DX

mov AX,[SI]

M_1: imul X

jo Mes1

inc SI

inc SI

add AX, [SI]

jo Mes1

loop M_1

mov Y, AX

```

cmp AX, 0
jge Mes3
mov DX, offset Mes_2      ;Вывод сообщения
mov AH, 09                ;на
int 21h                  ;экран.
Exit: mov AL, 0
mov AH, 4Ch
int 21h
Mes1: mov DX, offset Mes_1 ;Вывод сообщения
mov AH, 09                ;на
int 21h                  ;экран.
jmp Exit
Mes3: mov DX, offset Mes_3 ;Вывод сообщения
mov AH, 09                ;на
int 21h                  ;экран.
jmp Exit
END Start

```

Таблица 6.1

Исходные данные											
N = 4, X = 10											
№ варианта	a ₁	a ₂	a ₃	a ₄	a ₅	№ варианта	a ₁	a ₂	a ₃	a ₄	a ₅
1	-4	9	0	-5	0	9	5	-9	-9	9	4
2	0	-9	8	6	-9	10	3	6	6	7	6
3	2	0	9	6	6	11	-2	8	7	4	9
4	3	2	-9	2	8	12	0	-7	-9	3	-3
5	-3	9	5	1	2	13	-1	6	-4	-9	8
6	-2	8	-5	0	1	14	2	4	3	8	0
7	-3	7	6	-9	9	15	4	0	6	-8	-1
8	3	-7	6	9	-8						

4.4 Добавить в программу вывод на экран вида полинома в виде $Y = a_1 \cdot 10^4 + a_2 \cdot 10^3 + a_3 \cdot 10^2 + a_2 \cdot 10 + a_5$, подставив в выражение значения своего варианта задания.

4.5 Используя, описанный в разделе 2, алгоритм преобразования десятичного двоичнокодированного числа в символьный вид, разобрать исходный текст программы **CONV** (см. ниже). Создать и отладить исполняемый модуль программы **CONV**, выполнив этапы ассемблирования и компоновки. Добавить в исходный модуль программы недостающие комментарии. **Использовать упрощенные директивы сегментации и модель памяти ближнего типа.**

TITLE CONV

;Программа преобразования двоичнокодированного

;десятичного числа в символьный вид

Входные параметры:

; исходное число **Number**

;Выходные параметры:

; преобразованное в символьный вид число **Y_ASCII**

DATA SEGMENT ;Открыть сегмент данных.

Y_ASCII DB 7 DUP(?) ;Переменная для хранения символов ASCII.

Sign DB (?) ;Переменная для хранения знака числа.

Number DW 32500 ;Переменная для хранения исходного числа.

DATA ENDS ;Закрыть сегмент данных.

STK SEGMENT ;Открыть сегмент стека.

DB 100 DUP(?) ;Отвести под стек 100 байт.

STK ENDS ;Закрыть сегмент стека.

ASSUME DS:DATA, CS:CODE, SS:STK

CODE SEGMENT ;Открыть сегмент кодов.

PREOBR PROC ;Начать процедуру с именем PREOBR.

MOV AX, NUMBER ;Поместить в AX исходное число.

MOV SIGN, ' ' ;Поместить в переменную знака

;символ пробела (знак +).

CMP AX, 0; ;Сравнить число с нулем.

JNS M_1; ;Если больше или равно 0, перейти на

;метку M_1,

MOV SIGN, '-' ;иначе поместить в переменную знака

;символ минус (знак -).

NEG AX ;Преобразовать в прямой код.

M_1: XOR CX, CX ;Очистить CX.

MOV BX, 10 ;Поместить в BX делитель равный 10.

M_2: XOR DX, DX

DIV BX

PUSH DX

;Сохранить остаток в стеке.

INC CX

CMP AX, 0

JNE M_2 ;Если (AX) не равно 0, повторить деление.

XOR SI, SI ;Очистить SI.

MOV AL, SIGN ;Загрузить в AL знак числа.

MOV Y_ASCII[SI], AL ;Переслать знак в Y_ASCII.

```

        INC SI
M_3:  POP AX                ;Извлечь содержимое стека в AX.
        ADD AL, 30h         ;Вычислить ASCII код для цифры.
        MOV Y_ASCII[SI],AL ;Переслать ASCII код в Y_ASCII.
        INC SI
        LOOP M_3           ;Если содержимое CX не 0, повторить цикл.
        MOV Y_ASCII[SI], '$' ;Поместить символ конца строки в Y_ASCII.
        RET                ;Возврат из процедуры.
PREOBR ENDP                ;Завершить процедуру с именем PREOBR.

MAIN:  MOV AX, DATA        ;Основная программа.
        MOV DS, AX
        CALL PREOBR         ;Вызов процедуры преобразования
        MOV AX, 4C00h
        INT 21h
CODE  ENDS
END MAIN

```

4.6 Отредактировать исходный модуль программы **CONV**, используя в качестве переменной **NUMBER** вычисленное значение полинома для своего варианта задания. Создать и отладить исполняемый модуль программы **CONV_X** (**X** – номер варианта), выполнив этапы ассемблирования и компоновки.

4.7 Используя отлаженные программы **POLY_X** и **CONV_X** в виде процедур, составить программу **PROG_X**, которая вычисляет полином, выводит его вид и его рассчитанное значение на экран монитора.

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также фамилии И.О. студента, подготовившего отчёт;
- цель работы;
- вариант задания;
- листинги программы **POLY_X** и **CONV_X** с комментариями;
- листинг программы **PROG_X**;
- результат работы программ **POLY_X**, **CONV_X** и **PROG_X**.

6 Контрольные вопросы

- 6.1 Команды арифметических операций. Формат, операнды и флаги?
- 6.2 С какой целью используются подпрограммы и процедуры?
- 6.3 Как выглядит типовая структура для организации процедуры?

- 6.4 Как выглядит типовая структура для организации подпрограммы?
- 6.5 Какие действия выполняются в микропроцессоре при выполнении команды вызова **CALL** и команды возврата **RET**?
- 6.6 В каком месте программы могут располагаться процедуры?
- 6.7 Как можно организовать передачу входных параметров процедуры?
- 6.8 Что такое процедура-функция?
- 6.9 Какими командами можно активизировать арифметические флаги микропроцессора?
- 6.10 С помощью какого набора команд можно вывести сообщение на экран?
- 6.11 Объясните разницу между числом, цифрой и символом цифры. Что представляет собой таблица ASCII кодов?
- 6.12 В чем заключается суть преобразования десятичного двоично-кодированного числа в символьный вид?
- 6.13 Укажите диапазон беззнаковых чисел размером **BYTE**, **WORD**, **DOUBLE WORD**.
- 6.14 Укажите диапазон знаковых чисел размером **BYTE**, **WORD**, **DOUBLE WORD**.
- 6.15 В каком виде могут храниться числовые данные в памяти и в регистрах микропроцессора?

7 Рекомендуемая литература

- 7.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с. 165...181.
- 7.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 50...54, 82...86.
- 7.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К. Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 56...60, 91...96.

Лабораторная работа №7

Исследование организации переходов и циклов

1 Цель работы

Изучение команд передачи управления и получение практических навыков отладки разветвляющихся программ.

2 Теоретический материал

2.1 Кодировки символов

Вычислительная машина может обрабатывать не только числа, но и текстовую информацию, представляющую последовательность символов. Под символами подразумеваются буквы алфавита, цифры, знаки препинания, служебные символы и т. д. При этом каждому символу ставится в соответствие определенная двоичная кодовая комбинация. Соответствие между набором символов и их кодами называется кодировкой символов, а совокупность возможных символов и соответствующих им двоичных кодов образуют **таблицу кодировки**. В настоящее время существует множество таблиц кодировок. Общим для них является весовой принцип, согласно которому коды цифр возрастают с увеличением цифры, а коды букв увеличиваются в алфавитном порядке.

Наряду с **16-и битовой кодировкой Unicode**, которая используется с 1993 года и определяет универсальный набор символов (**UCS, Universal Character Set**), весьма популярными являются восьмиразрядные кодировки. Их недостаток, заключающийся в числе кодируемых символов не превышающим 256, заставил разработчиков создать несколько модификаций таблиц кодировок. Например, американский стандартный код для обмена информацией **ASCII (American Standard Code for nformation Interchange)** имеет европейские модификации (стандарт **ISO 8859**) для разных языков. Кодировка для русского языка представлена в таблице 7.3.

Коды всех символов этой таблицы представлены в десятичной (**d**) и шестнадцатеричной (**h**) системах. Первые 32 кода (1...32) имеют два значения: управляющие символы и изобразительные символы. Когда DOS пересылает эти коды на монитор или принтер, они выполняют управляющие функции, а не отображают символы, например, 8 – возврат на одну позицию, 9 – горизонтальная табуляция, 10 – перевод строки, 13 – возврат каретки, 32 – пробел. Для получения изображения символов эти коды необходимо занести в буфер экрана (начальный адрес 0B800:0000H).

Для определения кода какого-либо символа из таблицы следует

выполнить простые действия:

- для **шестнадцатеричных значений** – записать старшую и младшую шестнадцатеричные цифры, которые определяют положение клетки таблицы с нужным символом. Например: **Q = 51h**;
- для **десятичных значений** – найти сумму десятичных чисел, которые определяют положение клетки с нужным символом. Например: **Q = 80d + 1d = 81d**.

2.2 Команды передачи управления

Адрес следующей выполняемой микропроцессором команды определяется содержимым его регистров **CS:IP (EIP** в случае 32-х разрядной адресации). Если все команды находятся в одном сегменте кодов, то при переходе к следующей команде изменяется только содержимое **указателя команд IP**.

В линейных алгоритмах, когда все команды выполняются в том порядке, в котором они записаны, **содержимое IP** автоматически увеличивается на число, равное длине выполняемой команды.

В алгоритмах с ветвлением приходится изменять линейный ход программы. Для этого предназначены **команды передачи управления**. Они подразделяются на команды:

- безусловных переходов;
- условных переходов;
- вызовов;
- возвратов;
- управления циклами;
- прерываний.

При выполнении команд передачи управления флаги не устанавливаются.

Адрес команды, которой передается управление, в ассемблере задается с помощью **метки**, которая является символическим именем команды. С меткой связывается определенная ячейка памяти, в которой находится первый байт помеченной команды.

Команда безусловного перехода имеет мнемонику **JMP (JuMP)** и записывается в формате

JMP [модификатор] метка; перейти на метку.

В зависимости от того, где находится помеченная команда, машинные коды команд **JMP** различаются и имеют пять разновидностей:

- **прямой короткий переход (short)** – адрес перехода лежит в диапазоне **-128 ...+127**.

- **прямой ближний переход (near)** – помеченная команда находится в текущем сегменте кода на расстоянии **128 ...2¹⁶** адресов от команды JMP;
- **прямой косвенный ближний переход** – адрес перехода задается косвенно с помощью ссылки на регистр или ячейку памяти в которых он находится;
- **прямой дальний переход (far)**– помеченная команда находится в другом сегменте кодов. Изменяется содержимое регистров CS:IP (EIP);
- **прямой косвенный дальний переход** – полный адрес перехода **CS:IP (EIP)** содержится в ячейках памяти.

Иногда в программах имеет смысл указывать тип перехода с помощью **модификатора**:

- **short** – прямой короткий переход;
- **near ptr** – прямой ближний переход;
- **far ptr** – прямой дальний переход;
- **wordptr** – косвенный ближний переход;
- **dword ptr** – косвенный дальний переход.

Примеры:

```
JMP m_1;           ;перейти на метку m_1
JMP near ptr m_1;   ;перейти на метку m_1
MOV BX, offset m_2; ;загрузить в BX адрес ближнего перехода
JMP BX;             ;перейти на метку адрес которой находится в BX
```

Команды условных переходов организуют передачу управления на метку в случае выполнения заданного условия. **Если заданное условие не выполняется, то происходит переход к следующей по порядку команде.** Команды не влияют на формирование флагов. Обобщенный формат команд условного перехода имеет вид:

Jсond метка;

Jump - прыжок, **Condition** – условие, **метка** – метка перехода, которая может находиться только в **текущем сегменте кода**. Отсюда следует, что **все условные переходы являются короткими или ближними** внутрисегментными.

Условия, на основании которых формируется решение о переходе, определяются состоянием флагов микропроцессора. Поэтому при использовании команд условного перехода необходимо следить за тем, чтобы флаги находились в активном состоянии. Активизиру-

вать флаги можно выполнением «безобидной» арифметической команды, либо команды **сравнения CMP**.

Команды **условных переходов по флагам** используют в качестве условия один из арифметических флагов результата операции:

JC метка	;перейти на метку, если CF = 1
JNC метка	;перейти на метку, если CF = 0
JP метка	;перейти на метку, если PF = 1
JNP метка	;перейти на метку, если PF = 0
JZ метка	;перейти на метку, если ZF = 1
JNZ метка	;перейти на метку, если ZF = 0
JS метка	;перейти на метку, если SF = 1
JNS метка	;перейти на метку, если SF = 0
JO метка	;перейти на метку, если OF = 1
JNO метка	;перейти на метку, если OF = 0

В мнемоническом обозначении команды второй или третий символ обозначают один из арифметических флагов, **единичное** значение которого используется в качестве условия. Символ **N** - Not означает, что в качестве условия используется **нулевое** значение арифметического флага.

Команды условных переходов по результатам операции сравнения (см. таблицу 7.1) позволяют формировать условия перехода на основе анализа нескольких признаков (флагов), сформированных командой **сравнить CMP**. Такие команды принято делить на команды для анализа беззнаковых чисел (к ним применяется термины **выше (above)** / **ниже (below)**) и для анализа чисел со знаком (термины **больше (greater)** / **меньше (less)**). В случае равенства операндов используют термин **равно – equal**.

Примеры:

CMP AX, 0	;сравнить содержимое AX с 0
JE m_6	;если равно, перейти на метку m_6
CMP BX, 1024	;сравнить содержимое AX с числом 1024
JA m_2	;если выше, перейти на метку m_2.

2.3 Организация циклов в Ассемблере

Для организации многократного выполнения некоторого участка программы используется конструкция, называемая циклом. Для организации циклов необходимо выполнить следующую последовательность действий:

Таблица 7.1

Команды перехода по результатам сравнения

Типы операндов	Мнемоника	Условия перехода	Значения флагов
любые	JE	$op.1 = op.2$ (равно)	ZF = 1
любые	JNE	$op.1 \neq op.2$ (не равно)	ZF = 0
со знаком	JL/JNGE	$op.1 < op.2$ (меньше)	SF \neq OF
со знаком	JLE/JNG	$op.1 \leq op.2$ (меньше или равно)	SF \neq OF или ZF = 1
со знаком	JG/JNLE	$op.1 > op.2$ (больше)	SF = OF и ZF = 0
со знаком	JGE/JNL	$op.1 \geq op.2$ (больше или равно)	SF = OF
без знака	JB/JNAE	$op.1 < op.2$ (ниже)	CF = 1
без знака	JBE/JNA	$op.1 \leq op.2$ (ниже или равно)	CF = 1 или ZF = 1
без знака	JA/JNBE	$op.1 > op.2$ (выше)	CF = 0 и ZF = 0
без знака	JAЕ/JNB	$op.1 \geq op.2$ (выше или равно)	CF = 0

- 1) перед входом в цикл задать начальные значения переменных, которые изменяются в цикле (параметры цикла);
- 2) изменять эти переменные перед каждым новым повторением цикла;
- 3) пометить первую команду тела цикла меткой;
- 4) проверять условие окончания или повторения цикла;
- 5) управлять циклом, т. е. переходить к его началу или выйти из него по окончании.

В ассемблере для организации циклов используются специальные команды, относящиеся к командам передачи управления, которые позволяют организовать описанную выше последовательность действий простыми средствами. В качестве **счетчика числа повторений цикла** используется **регистр CX**, поэтому **максимальное число повторений составляет 2^{16}** . Перед началом цикла регистр CX инициализируется.

Для организации проверки условия окончания и управления циклом используется **команда LOOP метка** (повторить цикл). При выполнении этой команды микропроцессор производит следующие действия:

- содержимое счетчика циклов (регистра CX) **уменьшается на 1** (декремент CX);
- содержимое CX **сравнивается с «0»**;
- если **содержимое CX больше «0»**, осуществляется переход к началу цикла, который должен иметь **метку**. В противном случае выполняется выход из цикла.

Команда **LOOP** позволяет организовать только короткие переходы к началу цикла (в пределах $-128 \dots +127$ адресов) и это является ее недостатком. Поэтому для организации длинных циклов используются команды условных и безусловных переходов, которые были рассмотрены.

Команды LOOPE и LOOPZ позволяют повторять цикл пока содержимое регистра CX **не равно «0»** и **флаг нуля ZF** установлен в «1».

Команды LOOPNE и LOOPNZ позволяют повторять цикл пока содержимое регистра CX **не равно «0»** и **флаг нуля ZF** установлен в «0».

Эти команды отличаются от команды **LOOP** тем, что при их выполнении дополнительно анализируется флаг **нулевого результата ZF**. Это позволяет организовать досрочный выход из цикла, используя этот флаг в качестве индикатора.

2.4 Определение длины символьной переменной

При работе с символьными строками часто требуется знать их длину. Для вычисления длины символьной строки используется простой прием:

в **сегменте данных**, где объявлена строка, следует записать математическое выражение, которое при трансляции будет вычисляться и записываться в константу **length**:

.DATA

```
TEXT DB ' Turbo Assembler', 13, 10, '$' ;объявить строку
length = ($ - TEXT) - 3                  ;вычислить длину
                                          ;символьной строки
```

Поскольку переопределенный символ **\$** является счетчиком символов в строке, разность **(\$ - TEXT)** определяет длину символьной строки. Однако в объявленной строке присутствует на 3 символа больше: два управляющих символа **13 – перевод строки, 10 – возврат каретки** и завершающий строку символ **«\$»**. Поэтому найденную длину строки следует уменьшить на 3 байта.

2.5 Алгоритм программы CHANGE

Алгоритм программы замены строчных букв заглавными основан на анализе каждого символа исходной строки с целью определения принадлежности его к строчным или заглавным. Если символ строчный, то его код изменяется с помощью процедуры COR. Анализ происходит в цикле с числом повторений равным числу символов в строке.

Определение принадлежности символа к строчным или к заглавным происходит на основе таблицы 7.3 с **ASCII кодами**. Из нее следует, что строчные английские символы занимают диапазон кодов **61h...7Ah**. Поэтому коды символов, которые попадают в этот диапазон нужно корректировать. Коды символов не попадающих в этот диапазон корректировать не следует.

3 Подготовка к работе

- 3.1. Изучить методические указания и рекомендованную литературу.
- 3.2. Подготовить ответы на контрольные вопросы.

4 Задание на выполнение работы

- 4.1 Проанализировать приведенную ниже программу **CHANGE**. Создать и отладить исполняемый модуль программы **CHANGE**, выполнив этапы ассемблирования и компоновки. Добавить в исходный модуль программы недостающие комментарии.

TITLE CHANGE

;Программа заменяет строчные буквы заглавными в символьной

;строке и выводит на экран преобразованную строку на экран

;Входные параметры:

;текстовая переменная MYTEXT

.MODEL SMALL

.STACK 256

.DATA

MYTEXT DB 'Our NativeTown', 13, 10, '\$' ;объявляем текстовую
;переменную

;----- процедура коррекции кода символа -----

COR PROC NEAR

NOP

AND AH, 0DFh

MOV [BX], AH

RET

COR ENDP


```

;-----основная программа-----
Start:
    MOV AX, @DATA
    MOV DS, AX
    XOR AX, AX
    LEA BX, MYTEXT
    MOV CX, 15                ;загрузить счетчик циклов
;-----начало тела цикла-----
MT1: MOV AH, [BX]
    CMP AH, 61h
    JB MT2
    CMP AH, 7Ah
    JA MT2
    CALL COR
MT2: INC BX
;-----конец тела цикла-----
LOOP MT1                    ;повторить цикл, если (CX) ≠0
    LEA DX, MYTEXT          ;вывести переменную
    MOV AH, 09h              ;MYTEXT
    INT 21h                  ;на экран
    NOP                      ;холостая команда
    MOV AX, 4C00h            ;завершить
    INT 21h                  ;программу
END Start

```

4.2 Загрузите созданную программу в отладчик. В окне **Watches** введите имя символьной переменной **MYTEXT**. Произведите пошаговое выполнение, используя клавиши **F7** и **F8**. Обратите внимание, что использование **F7** позволяет пошагово выполнять команды внутри цикла, в то время как **F8** инициирует однократное выполнение цикла полностью. Наблюдайте и анализируйте результаты выполнения команд и изменения, происходящие с переменной в окне **Watches**.

4.3 Заново загрузите программу в отладчик. Клавишей **F2** установите ловушку (точку останова) на команде **NOP**, следующей после команд вывода символьной строки на экран. Запустите выполнение

программы клавишей **F9**. Программа должна выполняться до указанной точки останова.

4.4 Перейдите в окно **WINDOW** отладчика и пронаблюдайте результат выполнения программы, выбрав режим **USER SCREEN**. Повторным нажатием **F9** продолжите выполнение программы до ее окончания.

4.5 Внесите изменения в программу добавив автоматическое вычисление длины строки и загрузку вычисленным значением счетчика циклов **CX**.

4.6 Проверьте работу модифицированной программы **CHANGE** для другой символьной строки, содержащей не менее 20 строчных и заглавных букв английского алфавита.

4.7 Создайте и отладьте программу **CHANGE_1**, чтобы она обеспечивала выполнение задания в соответствии с вариантом из таблицы 7.2. Предусмотрите вывод на экран исходных и модифицированных строк текста.

Таблица 7.2

Варианты заданий

№ варианта	Заменить	№ варианта	Заменить
1	а) “а” на “А”, “s” на “S” б) все заглавные строчными	9	а) “1” на “А”, “2” на “В” б) все заглавные строчными
2	а) от “а” до “Г” на “А” до “F” б) все заглавные строчными	10	а) от “d” до “j” на “D” до “J” б) все заглавные строчными
3	а) “b” и “c” заглавными б) все заглавные строчными	11	а) “D” на “J”, “d” на “j” б) все заглавные строчными
4	а) от “Г” до “Z” заглавными б) все заглавные строчными	12	а) “3” на “C”, “4” на “D” б) все заглавные строчными
5	а) символ “ (” на “) ” б) все заглавные строчными	13	а) все пробелы на “!” б) все заглавные строчными
6	а) “y” на “Y”, “z” на “Z” б) все заглавные строчными	14	а) все пробелы на “=” б) все заглавные строчными
7	а) “o” на “O”, “r” на “R” б) все заглавные строчными	15	а) “s” на “S”, “t” на “T” б) все заглавные строчными
8	а) “Y” на “y”, “Z” на “z” б) все заглавные строчными		

5 Требования к отчёту

Отчёт должен содержать:

- титульный лист с указанием названия ВУЗа, кафедры, номера и темы лабораторной работы, а также фамилии И.О. студента, подготовившего отчёт;
- цель работы;
- вариант задания;
- листинги программы **CHANGE** и **CHANGE_1** с комментариями;
- результат работы программы **CHANGE** и **CHANGE_1**.

6 Контрольные вопросы

- 6.1 Назовите три типа команды безусловного перехода.
- 6.2 Какой может быть длина перехода в разных типах команды JMP?
- 6.3 Содержимое каких регистров модифицируется при выполнении безусловных переходов разных типов?
- 6.4 Какова максимальная длина условного перехода?
- 6.5 Каким образом может быть указан адрес перехода?
- 6.6 Какие флаги могут быть использованы в командах условного перехода после выполнения команды сложения?
- 6.7 Приведите возможные команды условных переходов, если после сравнения беззнаковых чисел D1 и D2 оказалось: а) $D1=D2$, б) $D1<D2$, в) $D1>D2$.
- 6.8 Приведите возможные команды условных переходов, если после сравнения чисел со знаками P1 и P2 оказалось: а) $P1\neq P2$, б) $P1<P2$, в) $P1\geq P2$.
- 6.9 Какие команды могут использоваться для организации циклов?
- 6.10 Какова максимальная длина переходов при организации циклов?
- 6.11 Какие признаки, кроме CX=0, могут быть использованы при организации циклов?
- 6.12 Как микропроцессор выполняет команду LOOP?
- 6.13 Как осуществляется переход к процедурам разных типов?
- 6.14 Назовите варианты команды возврата из процедуры.
- 6.15 В чем состоит разница кодов строчных и заглавных символов английского алфавита?
- 6.16 Каким образом можно заменить код строчной буквы на код заглавной?
- 6.17 Каким образом можно заменить код заглавной буквы на код строчной?
- 6.18 Как можно автоматически вычислять длину символьной строки?
- 6.19 Какие правила следует соблюдать при организации циклов на Ассемблере?

7 Рекомендуемая литература

- 7.1 Юров, В. И. Assembler [Текст]: учеб. пособие для вузов / В. И. Юров. 2-е изд. – СПб.: Питер, 2007. с. 209...235.
- 7.2 Финогенов, К. Г. Основы языка Ассемблера [Текст] / К. Г. Финогенов. – М.: Радио и связь, 2000. – с. 74...82.
- 7.3 Финогенов, К. Г. Использование языка Ассемблера [Текст]: учеб. пособие для вузов / К. Г. Финогенов. – М.: Горячая линия-Телеком, 2004. – с. 82...91.

Таблица 7.3

Таблица кодов ASCII

старшая часть кода																		
младшая часть кода		d	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
	d	h	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	0		►		0	@	P	'	p	A	P	a	▒	L	Ш	p	Ё
	1	1	☺	◄	!	1	A	Q	a	q	Б	С	б	▒	┐	т	с	ё
	2	2	☹	↕	"	2	B	R	b	r	В	Т	в	▒	└	т	т	ё
	3	3	♥	!!	#	3	C	S	c	s	Г	У	г		┌	Ш	у	ё
	4	4	♦	¶	\$	4	D	T	d	t	Д	Ф	д	└	—	Е	ф	ї
	5	5	♣	§	%	5	E	U	e	u	Е	Х	е	└	+	Р	х	ї
	6	6	♠	—	&	6	F	V	f	v	Ж	Ц	ж	└	└	Р	ц	ў
	7	7	•	↕	'	7	G	W	g	w	З	Ч	з	└	└	Р	ч	ў
	8	8		↑	(8	H	X	h	x	И	Ш	и	└	└	Р	ш	°
	9	9	○	↓)	9	I	Y	i	y	Й	Щ	й	└	└	Р	щ	•
	10	A	◼	→	*	:	J	Z	j	z	К	Ъ	к	└	└	Р	ъ	·
	11	B	♂	←	+	;	K	[k	{	Л	Ы	л	└	└	▀	ы	√
	12	C	♀	└	,	<	L	\	l		М	Ь	м	└	└	▀	ь	№
	13	D	♪	↔	-	=	M]	m	}	Н	Э	н	└	=	▀	э	я
	14	E	♫	▲	.	>	N	^	n	~	О	Ю	о	└	└	▀	ю	■
	15	F	☼	▼	/	?	O	_	o		П	Я	п	└	└	▀	я	

Краткая система команд микропроцессора i80X86

Команды передачи данных

MOV	dst, src	; dst:= (src)
XCHG	op1, op2	; op1:= (op2) ; op2:= (op1)
LEA	reg, mem	; reg:= [mem]
PUSH	src	; SP:= (SP)–2 ; [(SS):(SP)]:= (src)
POP	dst	; dst:= [(SS):(SP)]; SP:= (SP)+2

Команды арифметических операций

ADD	dst, src	; dst:= (dst)+(src)
ADC	dst, src	; dst:= (dst)+(src)+CF
SUB	dst, src	; dst:= (dst) – (src)
SBB	dst, src	; dst:= (dst) – (src) –CF
MUL	src	; AX:= (AL)* (src), DX:AX:= (AX)* (src), ; умножение байтов или слов без знака
IMUL	src	; умножение для чисел со знаками
DIV	src	; целочисленное деление беззнаковых чисел ; AL := quot ((AX)/(src)); частное при ; делении на байт. AH := rem ((AX)/(src)); остаток при ; делении на байт. AX := quot ((DX:AX)/(src)); частное ; при делении на слово DX := rem ((DX:AX)/(src)); остаток ; при делении на слово.
IDIV	src	; целочисленное деление чисел со знаками
CDW		; преобразование байта в AL в слово в AX DB → DW
CWD		; преобразование слова в AX в двойное слово в DX:AX DW → DD
CMP	op1, op2	; сравнение операндов (op1) –(op2)
INC	op	; op:= (op) + 1
DEC	op	; op:= (op) – 1
NEG	op	; op:= –(op)

Команды логических операций

OR	dst, src	; dst := (dst) v (src)
XOR	dst, src	; dst := (dst)⊕(src)
NOT	op	; инверсия op
AND	dst, src	; dst := (dst)^(src)
TEST	dst, src	; флаги:= (dst)^(src)

Команды сдвигов

SHL	op, N	; логический влево
SAL	op, N	; арифметический влево
SHR	op, N	; логический вправо
SAR	op, N	; арифметический вправо
ROL	op, N	; циклический влево
ROR	op, N	; циклический вправо
RCL	op, N	; циклический влево через перенос
RCR	op, N	; циклический вправо через перенос

Примечание: N=1 или содержимому регистра CL

Команды передачи управления

JMP	op	; безусловный переход, op – метка или ; адрес в регистре или ячейка памяти
-----	----	---

Команды условных переходов по флагам

Jcond метка ; условный переход к метке по флагу (признаку):
cond = C (CF=1), NC (CF=0), S (SF=1), NS (SF=0), Z (ZF=1), NZ (ZF=0),
 O (OF=1), NO (OF=0), P (PF=1), NP (PF=0).

Команды условных переходов по результатам операции сравнения

JE	метка	; op1 = op2 для любых чисел
JNE	метка	; op1 ≠ op2 для любых чисел
<i>Для чисел без знака Для чисел со знаком</i>		
JB / JNAE	метка;	JL / JNGE метка ; при op1 < op2
JBE / JNA	метка;	JLE / JNG метка ; при op1 ≤ op2
JA / JNBE	метка;	JG / JNLE метка ; при op1 > op2
JAE / JNB	метка;	JGE / JNL метка ; при op1 ≥ op2

Команды организации циклов

LOOP	метка	; CX := (CX) – 1, переход к метке при ; (CX) ≠ 0
LOOPZ / LOOPE	метка	; CX := (CX) – 1, переход к метке при ; (CX) ≠ 0 и ZF = 1
LOOPNZ / LOOPNE	метка	; CX := (CX) – 1, переход к метке при ; CX ≠ 0 и ZF = 0
JCXZ	метка	; переход к метке при (CX) = 0

Команды управления флагами

CLD	; DF ← 0	CLC	; CF ← 0
STD	; DF ← 1	STC	; CF ← 1
CLI	; IF ← 0	CMC	; инверсия CF
STI	; IF ← 1		

