

Министерство образования и науки РФ  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Ярославский государственный технический университет»  
Кафедра «Информационные системы и технологии»

**СБОРНИК ЛАБОРАТОРНЫХ РАБОТ  
ПО ДИСЦИПЛИНЕ  
«МАШИНОЗАВИСИМЫЕ ЯЗЫКИ  
ПРОГРАММИРОВАНИЯ»**

Ярославль  
2020

Царев Ю.В. Сборник лабораторных работ по дисциплине «Машинозависимые языки программирования» Ярославский гос. техн. ун-т. - Ярославль, 2020.- 72 с.

Учебное пособие представляет лабораторный практикум по дисциплине «Машинозависимые языки программирования», входящей в цикл предметов части ООП по направлению подготовки 090304 «Программная инженерия» профиль «*Управление проектами разработки программного обеспечения*».

Содержатся сведения, касающиеся компиляции, редактирования и отладки - этапов создания программ на машинно-ориентированном языке низкого уровня. Излагаются теоретические сведения об архитектурных особенностях микропроцессоров фирмы Intel, командах и директивах языка ассемблера. Приводятся описания машинных команд и примеры их применения в алгоритмах обработки информации. Теоретические материалы сопровождаются примерами практических заданий. В приложении приводится справочный материал по командам языка ассемблера.

## Содержание

Содержание .....	3
ВВЕДЕНИЕ .....	4
Лабораторная работа № 1 .....	5
Трансляция, компоновка и отладка программ .....	5
Лабораторная работа № 2 .....	21
Режимы адресации .....	21
Лабораторная работа № 3 .....	32
Программирование ветвлений и циклов .....	32
Лабораторная работа № 4 .....	44
Арифметические операции целочисленной обработки информации .....	44
Лабораторная работа № 5 .....	55
Программирование операций ввода-вывода .....	55
Приложение 1 .....	65
Программная модель микропроцессора Intel (Pentium III) .....	65
Приложение 2 .....	66
Система команд микропроцессора Intel 8086 .....	66
Выбор значения из таблицы .....	68
Приложение 3 .....	69
Формат команд передачи управления .....	69
Приложение 4 .....	70
Формат арифметических команд .....	70
Приложение 5 .....	71
Коды ASCII (диапазон 0-127) .....	71

## ВВЕДЕНИЕ

Компьютер стал повседневным рабочим инструментом людей множества профессий, многие из которых мало что знают о самом компьютере – стоит себе ящик, работает, когда его включишь, - чего еще надо? Однако такое положение вещей не должно устраивать студентов, которые выбрали вычислительную технику и информационные технологии своей профессией. Профессионалы в области информатики и вычислительной техники должны уметь подключать к компьютеру нестандартное оборудование, оптимизировать программные коды, защищать свои программы от несанкционированного доступа, - иначе говоря, работать с конкретным программно-аппаратным комплексом, осознавая все его достоинства и недостатки. Освоение этих премудростей возможно на основе знаний о внутреннем устройстве компьютера и о символическом представлении машинного языка – ассемблере. Предлагаемые методические указания к лабораторным работам по языку ассемблера предназначены помочь студентам в этом освоении.

Настоящие методические указания являются первой, начальной частью серии таких работ, имеющих целью углубленное изучение студентами команд компьютеров на базе микропроцессоров Intel и получение знаний об их архитектурных особенностях. Эти указания помогут Вам подготовить текст простой исходной программы преобразования данных, получить соответствующий исполнимый модуль и отладить его.

Для выполнения лабораторных работ необходимо аппаратное обеспечение в виде IBM- совместимого компьютера и программное обеспечение – редактор текстов, компилятор ассемблера, загрузчик и отладчик. Более подробно программное обеспечение описывается в теоретическом материале к первой лабораторной работе.

# **Лабораторная работа № 1**

## **Трансляция, компоновка и отладка программ**

### **1. ЦЕЛЬ РАБОТЫ**

Целью работы является освоение инструментальных средств создания программ на языке ассемблера.

### **2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

#### **2.1. Архитектура компьютера**

Структурные схемы компьютеров мало чем отличаются у разных моделей компьютеров. У всех компьютеров есть оперативная память, процессор, внешние устройства. Различными являются способы, средства и используемые ресурсы, с помощью которых компьютер функционирует как единый механизм. Совокупность функциональных программно-управляемых свойств компьютера называют архитектурой.

Архитектура ЭВМ – это абстрактное представление ЭВМ, которое отражает структурную, схемотехническую и логическую организацию. Приступить к изучению языка ассемблера любого компьютера имеет смысл только после выяснения того, какая часть из вышеперечисленных аспектов архитектуры оставлена видимой и доступной для программирования. Это так называемая программная модель компьютера. В приложении 1 приведена программная модель микропроцессора Intel (Pentium III) [1]. Как видно из приведенной модели, ее основу составляют регистры общего назначения, состояния и ряд других. В разных моделях компьютеров одной и той же фирмы-производителя количество и разрядность регистров могут существенно изменяться. Но в качестве базовой модели для языка программирования микропроцессоров Intel используется 14-регистровая система, на которой остановимся далее подробнее.

**Универсальные регистры AX, BX, CX, DX** имеют разрядность 16 бит (2 байта) и могут использоваться для временного хранения любых данных, при этом можно работать с каждым регистром целиком, а можно отдельно с каждым его байтом. Старшие байты РОН имеют имена AH, BH, DH, CH, а

младшие – AL, BL, DL, CL. Регистры AL, AH образуют соответственно младший и старший байты условного регистра AX.

Кроме того, каждый из РОН может использоваться как специальный регистр при выполнении некоторых команд программы:

регистр **AX**, аккумулятор, используется при умножении и делении слов, в операциях ввода-вывода и в некоторых операциях над строками;

регистр **AL** используется при выполнении аналогичных операций над байтами, а также при преобразовании десятичных чисел и выполнении над ними арифметических операций;

регистр **AH** используется при умножении и делении байтов;

регистр **BX**, базовый регистр, часто используется при адресации данных в памяти;

регистр **CX**, счетчик, используется как счетчик числа повторений цикла и в качестве номера позиции элемента данных при операциях над строками. Регистр CL используется как счетчик при операциях сдвига и циклического сдвига на несколько битов;

регистр **DX**, регистр данных, используется при умножении и делении слов. Кроме этого используется в операциях ввода-вывода как номер порта.

В микропроцессорах Intel программы и данные хранятся в отдельных областях памяти - сегментах, с объемом до 64 КБ (килобайт). Одновременно микропроцессор может иметь дело с 4 сегментами, начальные адреса которых содержатся в **сегментных регистрах** CS, DS, SS, ES. Эти регистры выполняют следующие функции:

- регистр сегмента команд **CS** указывает на сегмент, содержащий текущую исполняемую программу. Для вычисления адреса следующей подлежащей исполнению команды процессор складывает значение CS умноженное на 16 с указателем команд IP;

- регистр сегмента стека **SS** указывает на текущий сегмент стека - области памяти предназначенной для временного хранения данных и адресов;

- регистр сегмента данных **DS** указывает на текущий сегмент данных, который обычно содержит используемые в программе переменные;

- регистр дополнительного сегмента **ES** указывает на текущий дополнительный сегмент, который используется при выполнении операций над строками.

**Регистры смещений IP, SP, BP, SI, DI** используются для хранения относительных адресов ячеек памяти внутри сегментов (иначе говоря, смещений относительно начала сегментов):

- регистр **IP** хранит смещение адреса текущей команды программы;

- регистр **SP** указывает на вершину стека – смещение относительно начала стека;

- в регистр **BP** записывается начальный адрес поля памяти, непосредственно отведенного под стек;

- регистры **SI** и **DI** предназначены для хранения адресов индексов источника и приемника данных при операциях над строками и другими структурами данных.

**Регистр флагов FL** представляет собой 16-битовый регистр, где фиксируется информация о текущем состоянии процессора.

15

0

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
--	--	--	--	----	----	----	----	----	----	--	----	--	----	--	----

Рисунок 1 – Регистр флагов

Флаг **OF** называется флагом переполнения и **OF=1** свидетельствует о наличии ошибки в операциях над числами со знаком.

Флаг направления **DF** используется в командах работы со строками. При **DF=1** регистр индекса, используемый в командах работы со строками, увеличивается на 1 при каждом следующем выполнении команды, при **DF=0** – регистр индекса на 1 уменьшается.

Флаг прерывания **IF** обычно он устанавливается в 1 и такое его значение позволяет исполняемой программе пользователя реагировать на прерывания. Однако, когда вызывается программа обработки прерывания,

флаг IF устанавливается в 0, чтобы никакие другие прерывания не могли помешать текущей обработке прерывания.

Флаг TF называется флагом трассировки, при его значении, равном 1, разрешается выполнение программы по шагам.

Флаг знака SF устанавливается в 1, если в результате выполнения операции над числами со знаком, получается отрицательное число.

Флаг нуля ZF устанавливается в 1, если результатом операции является нулевое значение.

Флаг вспомогательного переноса AF используется в двоично-десятичной арифметике. Этот флаг устанавливается в 1, если результат такой операции не является десятичной цифрой.

Флаг четности PF устанавливается в 1, если результат операции имеет четное количество битов, равных 1, в двоичном представлении результата.

Флаг CF называется флагом переноса и в него заносится перенос (или заем) из знакового (старшего) разряда числа.

***Нужно уяснить, что не все команды программы на Ассемблере устанавливают в 0 или в 1 флаги. Причем, выполнение тех или иных команд связано с установкой конкретных флагов. Обратите внимание на это обстоятельство при изучении команд ассемблера.***

## **2.2. Процедуры формирования программы**

У большинства существующих реализаций ассемблера нет интегрированной среды, подобной Turbo Pascal или Turbo C. Поэтому для выполнения функций по вводу кода программы, ее трансляции, редактированию и отладке необходимо использовать отдельные служебные программы.

Последовательность процедур формирования программы на языке ассемблера и совокупность порождаемых файлов показана на рисунке 2.



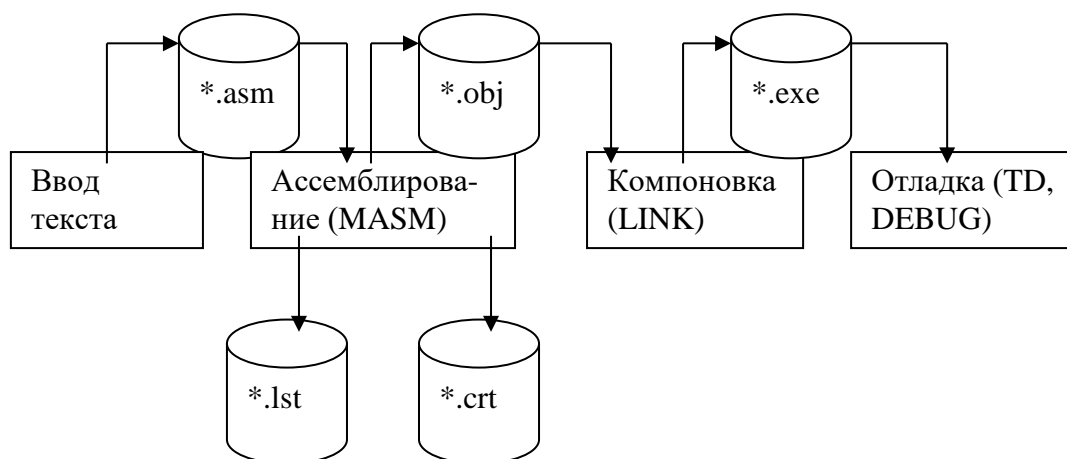


Рис. 2. Процесс обработки ассемблер-программы

В процессе формирования программы на языке ассемблера выделено 4 этапа:

- ввод исходного кода программы текстовым редактором,
- трансляция программы,
- создание загрузочного модуля,
- отладка программы.

Начальной процедурой создания программы на языке Ассемблера является ввод исходного текста программы в файл с расширением *.asm*. При этом может быть использован любой текстовый редактор, сохраняющий текст в виде стандартных кодов ASCII, например, редактор NC или блокнот. Основное требование к редактору, заключается в том, чтобы он (редактор) **не** вставлял посторонних символов (специальных символов форматирования).

Следующим шагом формирования программы является компиляция, которая носит специфическое название *ассемблирование*. Этот этап может быть выполнен программами ASM, MASM или TASM (сложность программ-компиляторов растет в указанной последовательности). Результатом выполнения этого этапа является программа в машинных кодах с расширением *.obj*, или, иначе, объектная программа, уже “понятная” микропроцессору. Естественно перевод состоится лишь в том случае, если исходный текст программы не содержит ошибок. Одновременно с объектным файлом могут быть созданы файлы листинга (*\*.lst*) и перекрестных ссылок

(*\*.crf*). Рекомендуется файл листинга создавать обязательно, поскольку при наличии ошибок в листинге описывается характер ошибки сразу после ошибочной команды, что значительно упрощает внесение исправлений, особенно на этапе обучения.

Файл листинга содержит код ассемблера исходной программы, машинный (объектный) код каждой команды и ее смещение в кодовом сегменте (значение регистра IP). Кроме того, сообщения о найденных синтаксических ошибках в программе помещаются непосредственно после ошибочной команды (бывают исключения, когда ошибка не в самой команде, а ранее нее, но эти ситуации встречаются редко). Строки в файле листинга имеют следующий формат:

<глубина\_вложенности>,            <номер\_строки>,            <смещение>,  
<машинный\_код>, <исходный код> ,

где **<глубина\_вложенности>** - уровень вхождения программного блока (программы, модуля, макроопределения, процедуры) в файл, **<номер\_строки>** - номер строки в файле листинга, он фигурирует в сообщениях об ошибках, но не обязательно совпадает с номером команды в исходном тексте, **<смещение>** - смещение в байтах текущей команды относительно начала сегмента кода, **<машинный\_код>** - машинное представление команды ассемблера, записанной правее в той же строке в поле **<исходный код>** , а **<исходный код>** является не чем иным, как записанной Вами командой языка ассемблер.

Однако объектная программа еще не является законченной и исполняемой, т.к. в ней определены не все адреса (программа не является “перемещаемой”) и не объединены части (блоки) программы, которые могут транслироваться отдельно с целью более простой отладки. Преобразование объектной программы в исполняемую (компоновка) выполняется загрузчиком (редактором связей) LINK либо TLINK (в зависимости от используемой программы ассемблирования: для ASM, MASM – LINK, для TASM – TLINK).

Чтобы проверить работоспособность созданной программы и увидеть результаты ее работы (если не использован вывод на дисплей), применяют программу отладчик. Тестирование и отладка исполняемой программы выполняется отладчиком TD или DEBUG.

Отладчик *td.exe*, разработанный фирмой Borland International представляет собой оконную среду отладки программ на уровне исходного текста на языках Pascal, C, ассемблер. Основные возможности отладчика, наиболее широко используемые студентами - это:

- выполнение трассировки программы в прямом направлении, при котором за 1 шаг выполняется одна машинная инструкция;
- просмотр и изменения состояния аппаратных ресурсов микропроцессора во время командного выполнения программы.

Управлять работой отладчика можно с помощью системы меню двух типов:

- глобального, находящегося в верхней части экрана и постоянно доступного. Вызов меню осуществляется нажатием клавиши F10;
- локального, учитывающего особенности окон и становящегося активным щелчком правой мыши или нажатием клавиш Alt+F10.

Специфика программы на ассемблере в том, что делать выводы о правильности ее функционирования можно, отслеживая работу на уровне микропроцессора, обращая внимание на то, как изменяется состояние ресурсов микропроцессора и компьютера в целом. Общее поведение программы позволяет просмотреть режим безусловного выполнения, который вызывается нажатием клавиши F9. Однако для детального изучения работы программы рекомендуется применять режим выполнения программы по шагам, для вызова которых выбираются пункты меню ***Run -> Trace into*** (прерывание или внутренняя процедура будут выполняться по шагам) или ***Run -> Step over*** (вызов процедуры или прерывание отрабатываются как одна обычная команда). При этом используется окно ***CPU***, вызов которого

осуществляется через глобальное меню командой **View -> CPU**. Окно **CPU** состоит из 5 подчиненных окон:

- окно с исходной программой в машинных кодах,
- *Register* – окно регистров микропроцессора, отражающее текущее состояние регистров,
- окно флагов, отражающее состояние флагов микропроцессора в соответствии с таблицей 1;

Таблица 1

Обозначения и значения флагов			
Имя флага	Обозначение флага	Установлен	Сброшен
Флаг переполнения	<b>O</b>	<b>1</b>	<b>0</b>
Флаг направления	<b>D</b>	<b>1</b>	<b>0</b>
Флаг прерывания	<b>I</b>	<b>1</b>	<b>0</b>
Флаг знака	<b>S</b>	<b>1(&lt;0)</b>	<b>0(&gt;0)</b>
Флаг нуля	<b>Z</b>	<b>1</b>	<b>0</b>
Флаг вспомогательного переноса	<b>A</b>	<b>1</b>	<b>0</b>
Флаг четности	<b>P</b>	<b>1</b>	<b>0</b>
Флаг переноса	<b>C</b>	<b>1</b>	<b>0</b>

- окно стека, в котором отражается содержимое области памяти, отведенной для стека,
- окно дампа оперативной памяти Dump, отражающее содержимое области памяти по адресу, указанному в левой части окна. В окне можно увидеть содержимое произвольной области памяти, для этого нужно в локальном меню, вызываемом по щелчку правой кнопки мыши, выбрать нужную команду.

Рекомендуемый порядок работы с отладчиком:

- а) вызвать на выполнение **td.exe.**;
- б) выбрать файл исполняемой программы, набрав комбинации клавиш **FILE ->OPEN** и имя Вашей программы в окне запроса. После ответа ОК на сообщение об отсутствии символьной таблицы в окно **CPU** загружается программа с нулевого адреса относительно начала сегментного регистра

кодов (для приведенного в конце описания лабораторной работы примера это будет команда PUSH DS);

в) выбрать режим пошагового выполнения **Run -> Step over**. В окне **CPU** появляется окрашенный треугольник между относительным адресом команды и машинным кодом команды. Он показывает очередную команду, которая будет выполнена процессором после нажатия функциональной клавиши F8. Изменения, которые происходят в сегментных регистрах после выполнения команды, отмечаются белым цветом соответствующей строки в окне регистров. Пошаговый процесс выполнять до тех пор, пока не появится сообщение об окончании программы (с ключевым словом *terminated*);

г) после выполнения команд, связанных с изменением содержимого ячеек памяти, нужно просматривать эти изменения командой **VIEW -> DUMP**. При отсутствии мыши скрыть окно дампа памяти можно нажатием функциональной клавиши F6.

### 2.3. Структура программы

Программа на языке ассемблера представляет собой последовательность операторов, описывающих выполняемые операции. Оператором (строкой) исходной программы может быть или команда, или псевдооператор (директива) ассемблера. Команды выполняются в процессе решения задачи на компьютере, а директивы – в процессе ассемблирования (трансляции) программы. Следовательно, в отличие от команд директивы сообщают ассемблеру (транслятору), что ему делать с командами и данными, которые вводятся в программе. Ниже в таблице 2 перечисляются наиболее часто используемые директивы ассемблера [5].

## Синтаксис и функции псевдооператоров (директив)

Псевдооператор	Формат и Функция
1	2
<u>Определения данных</u> DB	[имя] DB выражение [,.....] определяет переменную или присваивает ячейке памяти начальное значение. Резервирует 1 или более байт (при наличии запятых)
DW	[имя] DW выражение [,.....] аналогично предыдущему резервирует двухбайтовые слова
DD	[имя] DD выражение [,.....] Резервирует 4-х байтовые двойные слова
<u>Определения сегмента или процедуры</u> SEGMENT	Имя_seg SEGMENT [тип_выравнивания (подгонки)] [тип_связи] ['класс'] ..... Имя_seg ENDS Определяет границы сегмента программы. Обязательно содержит начало описания Имя_seg SEGMENT и окончание описания Имя_seg ENDS
ASSUME	ASSUME регистр_seg: Имя_seg [,.....] Или ASSUME регистр_seg: NOTHING Сообщает Ассемблеру, какой регистр сегмента связан с соответствующим сегментом программы. Оператор ASSUME регистр_seg: NOTHING отменяет действие всех предыдущих операторов ASSUME для данного регистра

1	2
PROC	Имя PROC [NEAR] или Имя PROC FAR  ....  .... RET имя ENDP Присваивает имя последовательности операторов. Должно иметь начало PROC и окончание ENDP
Псевдооператор	Формат и Функция
<u>Управление трансляцией</u> END	END [метка точки входа] Отмечает конец исходной программы
<u>Внешние ссылки</u> PUBLIC	PUBLIC идентификатор Делает определенный ранее идентификатор доступным другим модулям программы, которые впоследствии должны быть присоединены к данному модулю
EXTERN	EXTERN имя: тип [, ....] Указывает, что имя определено в другом модуле программы
INCLUDE	INCLUDE файл вставляет содержимое указанного файла в текущий файл исходной программы
<u>Определение идентификаторов</u> EQU	Имя EQU текст или Имя EQU числовое_выражение Постоянно присваивает идентификатору <b>имя</b> текст или числовое_выражение Имя = числовое_выражение Числовое_выражение присваивается идентификатору <b>имя</b> , но может быть переприсвоено

Обязательные требования к структуре ASM-программы следующие [6]:

- программа может использовать четыре сегмента памяти, начальные адреса которых должны быть загружены в регистры микропроцессора CS, DS, ES, SS, а сами сегменты в явном виде определены в программе в виде операторных скобок

- *имя\_сегмента segment*

- .....

- *имя\_сегмента ends,*

например,

```
DSEG          SEGMENT  PARA PUBLIC 'DATA'
SOURCE  DB    10,20,30,40
DEST    DB    4 DUP(?)
DSEG          ENDS;
```

- в программе должно быть указание, какие сегментные регистры закрепляются за используемыми именами регистров, например:

```
ASSUME  CS:CSEG, DS:DSEG, SS:STACK.
```

При исполнении программы адреса сегментных регистров CS, SS, ES в соответствии с вышеприведенными указаниями загружаются автоматически;

- сегмент данных DS в EXE-программе не может быть загружен автоматически, поскольку он используется программой-загрузчиком LINK для формирования начального адреса служебной области памяти – префикса программного сегмента (PSP), непосредственно предшествующего любой исполняемой программе. PSP – это группа служебных слов в оперативной памяти, формируемая для каждой загружаемой программы пользователя и занимающая обычно 256 байт (100H байт), именно адрес этой области записывается в регистр DS. Поэтому в самом начале исполняемой программы этот сегмент иницируется принудительно: сначала в стек записывается адрес возврата к служебной области в виде 2-ух слов – содержимого регистра DS и нулевого смещения; затем в регистр DS записывается адрес сегмента данных исполняемой программы, например, как показано ниже:

```
PUSH DS ; поместить в стек адрес PSP
```



```

SUB AX,AX           ; обнулить регистр AX
PUSH AX             ; поместить в стек смещение адреса возврата=0
MOV AX,DSEG         ; инициировать адрес сегмента данных
MOV DS,AX           ; загрузить адрес в регистр DS;

```

- в исходной программе обязательно должна быть определена метка для первой команды, с которой начнется выполнение программы. Это может быть собственно метка или имя процедуры, как показано в приведенной ниже программе. Имя этой метки обязательно должно быть указано в конце программы в качестве операнда директивы **END**, например,

```

END OUR_PROG;

```

- обеспечение выхода из программы, например, используя функцию 4C прерывания 21H, как показано ниже:

```

MOV AX,4C00H
INT 21H

```

или оформив основную программу как процедуру с атрибутом FAR и стандартным выходом из процедуры RET, как показано в приведенной ниже программе.

## 2.4. Пример программы на Ассемблере

```

TITLE      EX_PROG
           PAGE          ,132
STACK      SEGMENT PARA STACK 'STACK'
           DB 64 DUP('STACK ') ; Область стека
STACK      ENDS
DSEG       SEGMENT  PARA PUBLIC 'DATA'
SOURCE     DB 10, 20, 30, 40 ; эта таблица будет скопирована
DEST       DB 4 DUP(?)       ; в эту таблицу в обратном порядке
DSEG       ENDS
SUBTTL     ОСНОВНАЯ ПРОГРАММА

```

PAGE

CSEG                SEGMENT PARA PUBLIC 'CODE'

ASSUME            CS:CSEG, DS:DSEG, SS:STACK

OUR\_PROG        PROC FAR

;занести в стек такие начальные значения, чтобы программа

; могла вернуть управление отладчику

    PUSH DS    ; поместить в стек номер блока адреса возврата

    SUB AX,AX    ; обнулить регистр AX, тоже можно сделать

командой        MOV AX,0

    PUSH AX    ; поместить в стек значение адреса возврата=0

; инициировать адрес сегмента данных

    MOV AX,DSEG

    MOV DS,AX

; присвоить элементам таблицы DEST нулевые начальные значения

    MOV DEST,0                ; обнуление 1-ого байта

    MOV DEST+1,0              ; обнуление 2-ого байта

    MOV DEST+2,0              ; обнуление 3-его байта

    MOV DEST+3,0              ; обнуление 4-ого байта

; скопировать таблицу SOURCE в таблицу DEST в обратном порядке, в  
качестве промежуточной ячейки пересылки использовать регистр AL

    MOV AL, SOURCE

    MOV DEST+3,AL

    MOV AL, SOURCE+1

    MOV DEST+2,AL

    MOV AL, SOURCE+2

    MOV DEST+1,AL

    MOV AL, SOURCE+3

    MOV DEST,AL

    RET ; возврат управления отладчику db

OUR\_PROG        ENDP

CSEG

ENDS

END OUR\_PROG

### **3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

1. Набрать приведенную в тексте программу на ассемблере с использованием редактора текста.
2. Оттранслировать программу в объектный код.
3. Скомпоновать программу (получить исполнимый файл). Изучить листинг программы.
4. Провести отладку программы и проверить получаемые результаты.
5. Внести в программу следующие изменения: задать исходную таблицу SOURCE из 5 двухбайтовых шестнадцатеричных переменных и скопировать эту новую таблицу в DEST.
6. В сегменте данных определить переменные, заполнив их следующими значениями:
  - 5 байтов A, B, C, D, E;
  - 5 двухбайтовых слов AA, BB, CC, DD, EE;
  - 5 двойных слов AAAA, BBBB, CCCC, DDDD, EEEE;
7. Получить исполнимый файл программы с данными пункта 6 и изучить дампы памяти данных с целью выяснения механизма выравнивания.

### **4. СОДЕРЖАНИЕ ОТЧЕТА**

Отчет должен включать:

- а) титульный лист;
- б) формулировку цели работы;
- в) описание результатов выполнения пунктов 3-7:
  - листинги программ;
  - результаты выполнения программ;
- г) выводы, согласованные с целью работы.

## 5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие группы регистров выделяются в микропроцессоре и каковы особенности их использования?
2. Какую функцию в микропроцессоре выполняет регистр флагов?
3. Как используется регистр команд IP?
4. Какие шаги необходимо выполнить для получения из программы на языке ассемблера исполняемого модуля?
5. Прокомментируйте содержание листинга программы.
6. В каких окнах и в каком виде отображается состояние микропроцессора при отладке программ с применением отладчика *td.exe*?

## Лабораторная работа № 2

### Режимы адресации

#### 1. ЦЕЛЬ РАБОТЫ

Целью работы является разработка простой программы преобразования данных для приобретения практических навыков программирования на языке ассемблера и закрепления знаний по режимам адресации.

#### 2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

##### 2.1. Команды ассемблера

Программа на языке ассемблера представляет собой последовательность операторов, описывающих выполняемые действия. Оператором (строкой) исходной программы может быть или *команда*, или *псевдооператор* (директива) ассемблера.

*Команды* представляют краткую нотацию (запись) системы команд. В некоторых руководствах они называются машинными командами, т.к. именно они сообщают процессору, какие действия необходимо выполнять. В отличие от команд *псевдооператоры* сообщают (транслятору), что ему делать с командами и данными, которые вводятся в программу.

Команда может включать до 4-х полей следующего вида:

[метка:] мнемокод [операнд] [; комментарий]

Поскольку в [ ] указываются необязательные поля, следовательно, команда обязательно должна содержать мнемокод выполняемого действия. Поля могут набираться в любом месте строки, но отделяйте поля друг от друга хотя бы одним пробелом и, если хотите разобраться в своей программе по истечению времени, позаботьтесь о читабельности, что чаще всего обеспечивается за счет позиционирования полей. Пример команды со всеми полями:

GETCOUNT:    MOV        CX, DX                    ;инициализация  
счетчика, поместить содержимое регистра DX в регистр CX

Пример команды с полем мнемокода:

## PUSHF

Эта команда сохраняет в стеке содержимое регистра флагов.

Перечень команд ассемблера, применяемых во всех моделях микропроцессоров Intel приводится в приложении 2.

В качестве операнда в команде может фигурировать константа, которая может вводиться в следующих формах:

- а) двоичной, как последовательность цифр 0 и 1, заканчивающихся буквой B, например, 10111010B;
- б) десятичной, в привычной десятичной системе счисления с необязательной буквой D на конце, например, 129D или просто 129;
- в) шестнадцатеричной, как последовательность цифр от 0 до 9 и букв от A до F, заканчивающаяся буквой H. Если шестнадцатеричная константа начинается с буквы, то такая константа дополняется первым символом - цифрой от 0 до 9, например, 0E23H (в данном случае первая цифра информирует Ассемблер о том, что E23 число, а не идентификатор или переменная);
- г) литералом, в виде строки букв, цифр и других символов, заключенной в кавычки или апострофы.

Мнемокоды могут иметь от 2 до 6 букв, при трансляции мнемокод преобразуется в числовое значение по таблице перекодировки (внутри транслятора). Мнемокоды имеют жесткий формат, предусматривающий 1,2 или отсутствие операндов. Если операндов 2, они отделяются друг от друга запятой.

Нельзя использовать в качестве меток имена регистров и мнемокоды, кроме того метка должна начинаться с буквы, но может содержать цифры и специальные символы: ?, @/, \_, \$ и точку, однако точка может быть только первым символом метки.

Важной особенностью машинных команд является то, что они не могут манипулировать одновременно 2-мя операндами, находящимися в оперативной памяти (ОЗУ). Это означает, что в команде только 1 операнд

может указывать на ячейку ОЗУ, другой операнд должен быть либо регистром, либо непосредственным значением. По этой причине возможны следующие сочетания операндов в команде:

- а) регистр - регистр;
- б) регистр - память;
- в) память – регистр;
- г) регистр - непосредственный операнд;
- д) память - непосредственный операнд.

Для команд характерно, что при наличии двух операндов первый из них является приемником, а второй – источником. Результат операции сохраняется по первому адресу, вот почему первый операнд никогда не может быть непосредственным операндом или, иначе говоря, константой.

## **2.2. Режимы адресации**

Смещение, которое вычисляется операционным блоком для доступа к находящемуся в памяти операнду, называется исполнительным адресом операнда. Этот исполнительный адрес показывает, на каком расстоянии (в байтах) от начала сегмента располагается искомый операнд.

В зависимости от используемого режима адресации получение исполнительного адреса может заключаться только в извлечении его как составной части исполняемой команды, а могут потребоваться дополнительные операции сложения составной части команды с содержимым других регистров.

Различают адресацию операндов:

- непосредственную, которая заключается в указании в команде самого значения операнда, а не его адреса;
- прямую, предполагающую указание в команде непосредственно исполнительного адреса;
- косвенную, при которой в команде указывается адрес регистра или ячейки памяти, в которых хранится адрес операнда или его составляющие;

- ассоциативную (используется в ассоциативных запоминающих устройствах, на ней останавливаться не будем);

- неявную, когда адреса операндов в команде не указываются, а подразумеваются кодом операции.

**Непосредственная** адресация имеет место, если операнд-источник является константой или переменной, которой присвоено постоянное значение. Например:

MOV AX, 500    загружает значение 500 в РОН AX или

.....

K    EQU 1024

.....

MOV        CX, K                    загружает в РОН CX константу 1024, определенную идентификатором K.

Следует отметить, что непосредственный операнд может быть задан простым выражением, в котором константы или идентификаторы констант связаны арифметическими операциями +, -, \* или / (в таких выражениях не должно быть скобок). Например:

MOV AX, 156\*10H/2.

Следует помнить, что диапазон посылаемых чисел (значений непосредственного операнда) определяется вместимостью приемника - если это однобайтовый регистр (AH, AL, BL ...), то в него можно посылать беззнаковые числа в диапазоне от 0 до 255, знаковые – от -128 до 127.

**Прямая регистровая адресация** имеет место в командах, оперирующих с содержимым РОН или сегментных регистров в качестве одного или обоих операндов команды. Например, команда:

MOV DS, AX

копирует содержимое РОН AX в сегментный регистр DS, при этом содержимое регистра AX не изменяется.

При использовании этого вида адресации в программах необходимо следить, чтобы разрядности обоих регистров были одинаковы.



**Прямая адресация** ячеек ОП имеет несколько вариантов:

- **прямая обычная** характеризуется тем, что смещение является составной частью команды и не требует при формировании исполнительного адреса дополнительных регистров, иными словами  $A_{исп} = A_{смещ}$ . Обычно применяется, если операндом служит помеченная переменная, например:

**MOV AX, SOURCE**

загружает слово из ячейки памяти в регистр. При этом в памяти старший байт следует за младшим, а не предшествует ему. Это обусловлено тем, что в памяти ЭВМ старшая часть располагается в ячейках памяти со старшими адресами. Поэтому схема приведенной команды будет следующей:

01	00	SOU RCE	BB
02	00		AA
03	00	SOU RCE+2	

После выполнения вышеуказанной команды пересылки регистр AX будет содержать AX=AABB.

Примеры прямой обычной адресации вы можете видеть в программе, приведенной в приложении 1.А;

- **прямая с индексированием:**  $A_{исп} = A_{смещ} + A_{инд}$ , причем  $A_{инд}$  находится в индексном регистре, например:

- **MOV AX, SOURCE[SI];**

- **прямая с базированием:**  $A_{исп} = A_{смещ} + A_{базы}$ ,  $A_{базы}$  находится в базовом регистре, например:

- **MOV AX, SOURCE[BX].**

Такая адресация предназначена для доступа к данным с известным смещением относительно некоторого базового адреса, при этом исполнительный адрес получается путем сложения значения сдвига с содержимым регистров BX или BP. Например, таблица TABLE содержит

поля фамилии (FAM 20 байт), имени (NAME 15 байт) и адреса (PLACE 50 байт). Тогда командами

```
MOV      BX, 20
MOV      AL, TABLE[BX]
```

получим в регистре AL первый байт имени.

- **прямая с индексированием и базированием:**  $A_{исп} = A_{смещ} + A_{базы} + A_{инд}$ ,

например:

```
MOV AX, SOURCE[BX+SI].
```

Возможна и такая форма записи команды:

```
MOV      AX, NUMBER [BP][SI].
```

Существует 2 варианта косвенной адресации ячеек ОП:

- **косвенная обычная**, когда исполнительный адрес находится в регистре, например:

```
MOV      AX, [BX].
```

Исполнительный адрес операнда может находиться в любом из сегментных регистров, кроме регистра стека (в базовом регистре BX, регистре указателя базы BP или индексном регистре SI или DI). Косвенный регистровый операнд заключается в квадратные скобки, что означает "в качестве адреса брать содержимое того адреса, на который указывает заключенный в квадратные скобки регистр". Чтобы адрес-смещение переменной мог оказаться в РОН, используется команда пересылки следующего вида:

```
MOV      BX, offset SOURCE.
```

Функции этой команды заключаются в том, что смещение (offset) ячейки памяти с именем SOURCE помещается в РОН BX. Естественно, в программе эта команда должна предшествовать команде пересылки с косвенной адресацией.

Т.к. содержимое регистра легко изменить в ходе выполнения программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство

применяется для организации циклических вычислений и для работы со структурами данных типа таблиц и массивов;

- **косвенная с индексированием**: от предыдущей отличается тем, что исполнительный адрес берется в виде суммы адресов, находящихся в базовом и индексном регистрах:

MOV AX, [BX+SI].

Смешанная непосредственная адресация ячеек памяти имеет несколько вариантов:

- **непосредственная обычная**:

MOV AX, offset pole.

Здесь в качестве непосредственного операнда берется смещение адреса переменной pole;

- **непосредственная с индексированием**, когда в качестве исполнительного адреса операнда берется сумма значений индексного регистра и непосредственного смещения:

MOV AX, [SI+const],

причем смещение, обозначенное **const**, может быть задано числом, идентификатором константы, смещением адреса переменной (offset), или их комбинацией в виде простого выражения;

- **непосредственная с базированием**, в которой, в отличие от предыдущей адресации, фигурирует базовый, а не индексный регистр:

MOV AX, [BX+const].

Форма записи смещения относительно базы может быть любой из 3-ех нижеприведенных:

MOV AX, [BX]+4 ,

MOV AX, 4[BX] ,

MOV AX, [BX+4].

Это примечание относится и к форме записи команд с индексированием (предыдущий вид адресации);

- **непосредственная с базированием и индексированием** отличается тем, что для вычисления исполнительного адреса берется сумма базового и индексного регистра, к которым добавляется непосредственно фигурирующее в команде смещение:

MOV        AX, pole[BX+SI+const].

Адресация с базированием и индексированием очень полезна при работе с двумерными массивами и таблицами. В ней исполнительный адрес вычисляется как сумма значений базового регистра, индексного регистра и (возможно) сдвига. В случае двумерного массива базовый адрес может содержать начальный адрес массива, а значения сдвига и индексного регистра могут содержать смещения по строке и столбцу. Допустимыми форматами команд являются следующие записи:

MOV        AX, [BX+2+DI],

MOV        AX, [DI+BX+2],

MOV        AX, [BX+2][DI],

MOV        AX, [BX+2+DI].

### **3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

1. Изучить приведенный теоретический материал к лабораторной работе.

2. Используя прямую (первый вариант) и косвенную (второй вариант) адресацию, написать программы на ассемблере, выполняющие алгоритмы преобразований из п. 5 (по указанию преподавателя).

#### ***Примечания к кодированию программ:***

- *программа с использованием прямой адресации будет похожа на приведенную в предыдущей лабораторной работе программу, только непосредственные значения смещений в командах должны измениться в соответствии с заданным алгоритмом;*

- *для варианта косвенной адресации зашлите адреса начала массивов в базовый и индексный регистры, например командами:*

**MOV        BX, OFFSET SOURCE и**

***MOV        DI, OFFSET DEST,***

***и далее используйте регистры BX и DI для адресации элементов массивов.***

3. Оттранслировать программу в объектный код.

4. Провести отладку программы и проверить получаемые результаты.

#### **4. СОДЕРЖАНИЕ ОТЧЕТА.**

Отчет должен включать:

- титульный лист;
- описание цели работы;
- описание задания на лабораторную работу;
- - словесные пояснения к алгоритму решения задачи и схему программы (обращаю Ваше внимание на то, что схема программы одна, а реализаций этой схемы должно быть две – с применением прямой и косвенной типов адресации);
- листинги программ;
- результаты выполнения программ;
- выводы.

#### **5. ВАРИАНТЫ ЗАДАНИЙ**

1. Задать одномерный массив, состоящий из X элементов (X задается преподавателем из диапазона [7..10]). Заполнить массив константами. Переместить заданный массив в другую область памяти, поменяв местами элементы с четными и нечетными номерами (поставив каждый элемент с четным номером на место нечетного элемента и каждый элемент с нечетным номером – на место четного)

а) элементы массива – однобайтовые;

б) элементы массива – двухбайтовые;

2. Задать одномерный массив, состоящий из X элементов (X задается преподавателем из диапазона [7..10]). Заполнить массив константами. Переместить в другую область памяти элементы с нечетными номерами

а) элементы массива – однобайтовые;

б) элементы массива – двухбайтовые;

3. Задать одномерный массив, состоящий из  $X$  элементов ( $X$  задается преподавателем из диапазона  $[7..10]$ ). Заполнить массив константами. Переместить в другую область памяти элементы с четными номерами

а) элементы массива – однобайтовые;

б) элементы массива – двухбайтовые;

4. Задать одномерный массив, состоящий из  $X$  элементов ( $X$  задается преподавателем из диапазона  $[7..10]$ ). Заполнить массив константами. Создать новый одномерный массив, поместив в него на место элементов с четными номерами элементы заданного массива с нечетными номерами и обнулив элементы нового массива с нечетными номерами

а) элементы массива – однобайтовые;

б) элементы массива – двухбайтовые;

5. Задать одномерный массив, состоящий из  $X$  элементов ( $X$  задается преподавателем из диапазона  $[7..10]$ ). Заполнить массив константами. Создать новый одномерный массив, поместив в него на место элементов с нечетными номерами элементы заданного массива с нечетными номерами и заполнив элементы нового массива с четными номерами максимальными значениями констант

а) элементы массива – однобайтовые;

б) элементы массива – двухбайтовые.

## **6. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Каков синтаксис команд ассемблера?

2. Какие группы директив Вы знаете? Какие из директив обязательны в программе на ассемблере?

3. Что такое исполнительный адрес и как он используется при определении физического адреса операнда?

4. В чем заключается различие прямых и косвенных режимов адресации?

5. Как различить в командах ассемблера прямые и косвенные режимы адресации?

6. Какие режимы адресации Вы знаете?

## Лабораторная работа № 3

### Программирование ветвлений и циклов

#### 1 ЦЕЛЬ РАБОТЫ

Целью работы является закрепление знаний по командам условного и безусловного переходов и циклов на примере программ на языке ассемблера, а также приобретение навыков написания программ с циклами.

#### 2 ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

##### 2.1. Команды условного перехода

Команды передачи управления реализуют изменение естественного порядка выполнения команд программы. Их можно разделить на 3 подгруппы, описание которых приведено в приложении 3 [5]. В мнемонические обозначения команд условного перехода входят буквы, которые определяют условия в соответствии с таблицей 1 [1]. В этой таблице операнд “метка перехода” или “близкая метка” отражает тот факт, что метка помеченной команды должна находиться в пределах текущего сегмента кода и на относительном расстоянии от команды перехода  $>-128$  и  $< 127$  байтов. Ограничение  $(-128..127)$  байтов снято у процессоров, начиная с модели 80386, однако ограничение передачи управления в пределах текущего сегментного кода действует и в моделях PENTIUM.

Таблица 1

Мнемокоды команд условного перехода

Буква мнемокода	Условие	Тип операндов
E	равно	любые
N	Не равно	любые
G	больше	Числа со знаком
L	меньше	Числа со знаком
A	Выше, в смысле “больше”	Числа без знака
B	ниже, в смысле “меньше”	Числа без знака

Решение о том, куда будет передано управление командой условного перехода, принимается на основании условия. Источниками таких условий могут быть:



- любая команда, изменяющая состояние арифметических флагов (ниже эти флаги будут перечислены);

- команда сравнения CMP.

Формат команды CMP:

*CMP* приемник, источник или *CMP* операнд1, операнд2.

Эта команда осуществляет вычитание (операнд1 - операнд2) или (приемник – источник), однако результат никуда не записывается, а только устанавливает флаги в соответствие с таблицей 2 [5].

Таблица 2

Значения флагов, устанавливаемые командой CMP

Сравниваемые операнды	Флаги			
	OF	SF	ZF	CF
Операнды без знака				
Источник < приемник	Н	Н	0	0
Источник = приемник	Н	Н	1	0
Источник > приемник	Н	Н	0	1
Операнды со знаком				
Источник < приемник	0/1	0	0	Н
Источник = приемник	0	0	1	Н
Источник > приемник	0/1	1	0	Н

В этой таблице приняты следующие обозначения:

- “Н” означает, что ‘не имеет значения’ или иначе, на этот флаг операция не влияет;

- 0/1 означает, что флаг устанавливается или в 1 или в 0 в зависимости от значений операндов (отрицательные или положительные или разнознаковые операнды сравниваются).

Приведем еще одну таблицу 3 [1], в которой отражается действие команд условного перехода по значениям анализируемых этими командами флагов. В этой таблице через слеш ‘/’ перечисляются идентичные команды, действие которых совершенно одинаково, и применение конкретной из них зависит от пристрастий программиста. Наличие идентичных команд объясняется тем фактом, что если число\_1 >число\_2, то можно с уверенностью утверждать, что число\_1 не (меньше или равно) число\_2.

Таблица 3

## Логика команд условного перехода

Тип операндов	Мнемокод команды	Критерий перехода	Значения флагов для перехода
любые	JE	Операнд_1=операнд_2	ZF=1
Любые	JNE	Операнд_1<>операнд_2	ZF=0
Со знаком	JL/JNGE	Операнд_1<операнд_2	SF<>OF
Со знаком	JLE/JNG	Операнд_1<=операнд_2	SF<>OF или ZF=1
Со знаком	JG/JNLE	Операнд_1>операнд_2	SF=OF и ZF=0
Со знаком	JGE/JNL	Операнд_1>=операнд_2	SF=OF
Без знака	JB/JNAE	Операнд_1<операнд_2	CF=1
Без знака	JBE/JNA	Операнд_1<=операнд_2	CF=1 или ZF=1
Без знака	JA/JNBE	Операнд_1>операнд_2	CF=0 и ZF=0
Без знака	JAЕ/JNB	Операнд_1=>операнд_2	CF=0

**2.2. Команда безусловного перехода**

Безусловный переход в программе на ассемблере производится по команде JMP. Полный формат команды следующий:

JMP [модификатор] *адрес\_перехода*.

*Адрес перехода* может быть либо меткой, либо адресом области памяти, в которую предварительно помещен указатель перехода.

В системе команд микропроцессора существуют несколько кодов машинных команд безусловного перехода. Их различия определяются дальностью перехода и способом задания целевого адреса. Дальность перехода определяется местоположением операнда *адрес\_перехода*. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется внутрисегментным или близким, а во втором случае – межсегментным или дальним.

Внутрисегментный переход предполагает, что изменяется только содержимое регистра IP. Можно выделить три варианта внутрисегментного перехода: *прямой короткий, прямой, косвенный*.

**Прямой короткий внутрисегментный переход** применяется, когда расстояние от команды JMP до адреса перехода не более чем 127 байтов выше или ниже. В этом случае транслятор языка формирует машинную команду безусловного перехода длиной 2 байта: первый байт – код операции, второй байт – смещение. В коде операции заложена информация о том, что второй байт интерпретируется как смещение. Здесь нужно отметить одну особенность транслятора ассемблера – он является однократным, иными словами, машинный код программы получается за один просмотр команд от начала программы до ее окончания. В связи с этим обстоятельством, если безусловный переход должен происходить на адрес до команды JMP, то транслятор может легко вычислить смещение. Если же переход короткий, но на метку после команды JMP, то транслятору нужно подсказать, что он должен сформировать команду безусловного короткого перехода. С этой целью в команде JMP используется модификатор SHORT PTR (полностью - SHORT POINTER или короткий указатель):

JMP SHORT PTR M1

..... не более 35-40 команд

M1:

MOV AL, 34H.

**Прямой внутрисегментный переход** отличается от короткого тем, что длина машинной команды составляет 3 байта, в которой два последних байта интерпретируются как смещение. Нетрудно определить, что в этом варианте можно осуществлять переход в пределах 64 Кбайт памяти относительно следующей за JMP команды.

**Косвенный внутрисегментный переход** означает, что в команде JMP указывается не сам адрес перехода, а место, где этот адрес записан. Например:

LEA BX, M1

JMP BX

.....

```

M1:      MOV AL, 34H

Или

DSEG      SEGMENT  PARA PUBLIC  'DATA'
ADDR      DW  M1
.....

CSEG      SEGMENT  PARA PUBLIC  'CODE'
          ASSUME   CS:CSEG, DS:DSEG, SS:STACK
.....

          JMP ADDR.

```

В командах косвенного перехода рекомендуется применять модификатор NEAR, т.к. при косвенном переходе не всегда транслятору удастся определить, находится адрес перехода в текущем сегменте кода или нет.

Команда прямого межсегментного перехода имеет длину 5 байт, из которых 2 байта составляет смещение адреса перехода, а другие 2 байта – значение сегментной составляющей (CS) того кодового сегмента, где находится адрес перехода. Например:

```

SEG1      SEGMENT  PARA PUBLIC  'CODE'
ASSUME    CS:SEG1, DS:DSEG1, SS:STACK
.....

JMP FAR PTR M2
.....

M1        LABEL FAR
.....

SEG1 ENDS

SEG2      SEGMENT  PARA PUBLIC  'CODE'
ASSUME    CS:SEG2, DS:DSEG2, SS:STACK
.....

M2        LABEL FAR

          JMP M1.

```

Во втором случае FAR необязательно, но если модификатор примените, то ошибки не будет. Необязательность объясняется тем, что метка находится раньше команды перехода и транслятор может самостоятельно определить, что переход является межсегментным.

**Команда косвенного межсегментного перехода** в качестве операнда имеет адрес области памяти, в которой содержится смещение и сегментная часть целевого адреса перехода. Например:

```
DSEG          SEGMENT  PARA PUBLIC 'DATA'
ADDR  DD  M1
.....

CSEG          SEGMENT  PARA PUBLIC 'CODE'
          ASSUME  CS:CSEG, DS:DSEG, SS:STACK
.....

          JMP ADDR

CSEG          ENDS

CS1          SEGMENT  PARA PUBLIC 'CODE'
          ASSUME  CS:CS1, DS:DS1, SS:ST1

M1  LABEL FAR

          MOV     AX, BX

.....

CS1          ENDS.
```

Одним из вариантов рассматриваемого перехода является **косвенный регистровый межсегментный переход**. Адрес перехода в этом варианте указывается в регистре, что удобно при программировании динамических переходов, когда конкретный адрес перехода определяется в процессе выполнения программы и помещается в регистр:

```
DSEG          SEGMENT  PARA PUBLIC 'DATA'
ADDR  DD  M1
.....

CSEG          SEGMENT  PARA PUBLIC 'CODE'
```

```

                ASSUME    CS:CSEG, DS:DSEG, SS:STACK
. . . . .
                LEA BX, ADDR
JMP DWORD PTR [BX]
. . . . .
CSEG                ENDS
CS1                SEGMENT PARA PUBLIC 'CODE'
                ASSUME    CS:CS1, DS:DS1, SS:ST1
M1 LABEL FAR
                MOV      AX, BX
. . . . .
CS1                ENDS.

```

В двойное слово ADDR помещается смещение адреса и начала сегмента кода, включающего метку M1, в нашем случае, начало сегмента CS1.

Т.о. модификаторы SHORT PTR, NEAR PTR и WORD PTR применяют при организации внутрисегментных переходов, а FAR PTR и DWORD PTR – при межсегментных переходах.

### 2.3. Организация циклов

При организации циклов широко используются команды INC (инкремент) и DEC (декремент), что означает добавление или вычитание единицы из целого числа, помещенного в ячейку памяти, РОН или индексный регистр. Команды имеют формат:

```

INC операнд ,
DEC операнд.

```

Такую программную конструкцию как цикл можно реализовать, используя в программе операции инкремента, декремента, условного и безусловного переходов. Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики ассемблера предусмотрели специальные команды цикла, например:

## LOOP метка\_перехода.

Команда означает 'повторить цикл'. Выполнение команды заключается в следующем:

- вычитании 1 из регистра CX;
- сравнении регистра CX с нулем;
- если  $CX=0$ , то управление передается на следующую после LOOP команду, иначе осуществляется передача управления на метку\_перехода.

Другими командами цикла являются команды:

LOOPE/LOOPZ метка\_перехода,

которые означают “повторить цикл, пока  $CX \neq 0$  или  $ZF=0$ ”. Обе команды совершенно идентичны, поэтому использовать можно любую из них. Отличаются эти команды от предыдущей команды анализом окончания цикла:

- если  $CX > 0$  и  $ZF=1$ , управление передается на метку\_перехода, иначе если  $CX=0$  или  $ZF=0$ , то выполняется следующая после команды LOOPE/LOOPZ команда.

Еще одной модификацией являются команды цикла

LOOPNE/LOOPNZ метка\_перехода,

которые означают, “повторить цикл, пока  $CX \neq 0$  или  $ZF=1$ ”. Как и в предыдущем случае обе команды совершенно идентичны. В них анализ окончания цикла выполняется по следующему правилу:

- если  $CX > 0$  и  $ZF=0$ , управление передается на метку перехода, иначе если  $CX=0$  или  $ZF=1$ , то выполняется следующая после команды LOOPNE/LOOPNZ операция.

**Общая особенность команд цикла в том, что они используют регистр общего назначения CX как счетчик числа повторений цикла, поэтому при их использовании не забудьте до метки\_перехода послать в этот регистр нужное число – количество повторений цикла!**

Недостаток всех команд цикла в том, что они реализуют только короткие переходы. Для работы с длинными циклами используются комбинации команд условного перехода и безусловного перехода.

Приведем пример использования вышеописанных команд в контексте подсчета количества нулевых, положительных и отрицательных элементов вектора (одномерного массива), состоящего из однобайтовых чисел.

Описания переменных в сегменте данных могут быть следующими:

```
Mas      db    -1, 0, 3, -8, 0, 9, -6, 1, 2, -5 ; заданный вектор
Len_mas  = $-mas      ; количество элементов в векторе
Sch_0     db    0      ; счетчик нулевых элементов вектора
Sch_pol   db    0      ; счетчик положительных элементов вектора
Sch_otr   db    0      ; счетчик отрицательных элементов вектора.
```

Фрагмент сегмента кода для подсчета элементов может быть следующим:

```
Mov      cx, len_mas      ; инициализация счетчика цикла
Xor      si, si           ; инициализация индексного регистра
Cycl:    cmp  mas[si], 0    ; сравниваем элемент вектора с 0
          Jz   zero        ; нуль-элементы считаем в блоке zero
          Jg   pol         ; элементы > 0 считаем в блоке pol
          Inc  Sch_otr      ; увеличиваем счетчик элементов < 0
          Jmp  kon_cycl
Zero:    Inc  Sch_0         ; увеличиваем счетчик нулевых элементов
          Jmp  kon_cycl
pol:     Inc  Sch_pol       ; увеличиваем счетчик элементов > 0
kon_cycl: inc  si          ; переходим к следующему элементу вектора
loop     cycl ; завершаем цикл.
```

### **3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

1. Изучить приведенный теоретический материал к лабораторной работе.



2. Написать программы в соответствии с заданным преподавателем вариантом.

3. Оттранслировать программы в объектный код.

4. Провести отладку программ и проверить получаемые результаты.

#### **4. СОДЕРЖАНИЕ ОТЧЕТА**

Отчет должен включать:

- титульный лист;
- описание цели работы;
- описание задания на лабораторную работу;
- словесные пояснения к алгоритму решения задачи и схему программы;
- листинги программ;
- результаты выполнения программ;
- выводы.

#### **5. ВАРИАНТЫ ЗАДАНИЙ**

1. Преобразовать символьную строку заданной длины, изменив все строчные буквы латинского алфавита на прописные.

2. Определить, сколько цифровых и нецифровых символов присутствует в заданной символьной строке.

3. Определить, сколько символов кириллицы и латиницы присутствует в заданной символьной строке.

4. Определить, сколько знаков отношения (<,>=) присутствует в заданной символьной строке.

5. Преобразовать заданную символьную строку, изменив прописные буквы латиницы на их порядковые номера в алфавите.

***Примечание к кодированию заданий 1-5: при написании алгоритмов преобразований необходимо использовать таблицу кодов ASCII, которая приводится в приложении 5.***

6. Подсчитать количество положительных и отрицательных элементов в заданном векторе и определить, каких элементов в векторе больше

а) элементы вектора однобайтовые;

б) элементы вектора двухбайтовые.

7. Подсчитать количество нулевых и ненулевых элементов в заданном векторе и определить, каких элементов в векторе больше

а) элементы вектора однобайтовые;

б) элементы вектора двухбайтовые.

8. Подсчитать количество неотрицательных элементов в заданном двумерном массиве

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

9. Подсчитать количество неположительных элементов в заданном двумерном массиве

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

10. Подсчитать количество положительных и отрицательных элементов в заданном двумерном массиве и определить, каких элементов в нем больше

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

***Примечание к кодированию пунктов 8-10: в программе необходимо реализовать такую конструкцию, как “вложенные циклы”. Кроме того, понятие массива и индексации массива весьма условны, ибо в памяти ЭВМ элементы массива располагаются последовательно, строка за строкой, в результате чего физическая структура двумерного массива и вектора (одномерного массива) оказываются одинаковыми. Отличие двумерного массива и вектора заключается в интерпретации области памяти, отведенной этим структурам. Наращивание индекса элемента структуры определяется алгоритмом обработки.***

## **6. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Каков синтаксис команд условного перехода?
2. Какие флаги анализируют команды безусловного перехода?
3. Как формируется машинный код команды безусловного перехода ассемблера?
4. Что такое близкий и дальний переходы в ассемблере?
5. Как различить в командах прямой и косвенный переходы?
6. Какие действия выполняют команды цикла в ассемблере?
7. Какую команду необходимо предусмотреть перед меткой перехода для цикла?

## **Лабораторная работа № 4**

### **Арифметические операции целочисленной обработки информации**

#### **1. ЦЕЛЬ РАБОТЫ**

Целью работы является закрепление лекционного материала по командам арифметических операций на языке ассемблера и приобретение практических навыков реализации вычислительных алгоритмов.

#### **2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

##### **2.1. Десятичные числа**

Десятичные числа – специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой разрядов из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом двоично-десятичном коде (BCD – Binary Coded Decimal). Микропроцессор может хранить такие числа в двух форматах:

- упакованный формат – в байте 2 десятичные цифры, при этом старшая цифра занимает старшие 4 бита, диапазон представления чисел в одном байте составляет 00-99;

- неупакованный формат - в байте 1 цифра в 4 младших битах. Старшие биты все имеют нулевое значение и называются *зоной*.

Описываются BCD-числа неупакованные, как DB, а упакованные как DT. Цифры неупакованного числа перечисляются через “,”, а упакованного - как обычное десятичное число, например:

PER_1	DB	2,3,4,5,6,8,2	;неупакованное 2865432
PER_2	DT	9875645	;упакованное 9875645.

##### **2.2. Сложение и вычитание целых чисел**

Мнемокоды и форматы арифметических команд приведены в приложении 6 [5]. Как видно из таблицы арифметических команд, сложение целых чисел осуществляется командами:

ADD операнд1, операнд2 – (операнд1=операнд1+операнд2) или

ADD приемник, источник– (приемник = приемник + источник)

ADC операнд1, операнд2 – (операнд1=операнд1+операнд2 + значение *cf*) или

ADC приемник, источник – (приемник = приемник + источник + перенос).

Операндами могут быть как 8-ми битовые, так и 16-ти битовые двоичные числа. Если результат операции не помещается в приемник, микропроцессор фиксирует ситуацию переполнения, устанавливая в 1 флаг переноса. Действия программы в этой ситуации могут быть следующими:

- прекратить выполнение программы по переполнению разрядной сетки;
- увеличить разрядность операндов.

Рассмотрим 2-ой вариант. Положим, складываются 8-миразрядные операнды из РОН AL и BL:

ADD AL, BL ; результат может превысить диапазон

; представления 8-ми разрядных чисел

JNC M1 ; проверяем наличие переноса и при отсутствии

;переходим на M1

ADC AH,0 ; расширяем разрядную сетку, добавив в

;результат РОН AH, теперь расширенный

; результат сложения помещается в 16-битовом РОН AX

M1: .

В этом фрагменте программы можно было опустить команду условного перехода и сразу после ADD выполнить команду ADC. Следует учесть, что после метки M1 результат нужно выбирать из регистра AX.

Команды ADD и ADC могут воздействовать на 6 флагов:

- флаг переноса CF – CF=1, если результат сложения не помещается в операнде-приемнике, в противном случае CF=0;
- флаг четности PF=1, если результат имеет четное число битов со значением 1, в противном случае PF=0;

- вспомогательный флаг переноса  $AF=1$ , если результат сложения десятичных чисел требует коррекции;
- флаг нуля  $ZF=1$ , если результат сложения равен 0;
- флаг знака  $SF=1$ , если сумма отрицательна (старший бит числа со знаком равен 1), в противном случае  $SF=0$ ;
- флаг переполнения  $OF=1$ , если сложение чисел одного знака приводит к результату, который превышает диапазон допустимых значений приемника в дополнительном коде, а сам приемник при этом меняет знак.

Специальных команд сложения десятичных чисел нет. Объясняется это тем, что микропроцессор все операнды интерпретирует как двоичные числа и складывает их по правилам сложения двоичных чисел. Чтобы уяснить, как происходит сложение десятичных чисел, рассмотрим такой пример: сложим числа 26 и 55 в упакованном виде:  $(0010\ 0110) + (0101\ 0101) = (0111\ 1011)$ .

Полученный результат  $7B_{16}$  не является десятичным упакованным числом. Поэтому результат должен быть скорректирован для представления в десятичном виде. Для этих целей разработчики ассемблера предложили 2 операции: *AAA* – корректировка результата для представления в кодах ASCII и *DAA* – корректировка результата для представления в упакованном десятичном формате. Обе команды не имеют операндов и по умолчанию корректируют значение из регистра AL.

Команда *AAA* преобразует содержимое регистра AL в правильную неупакованную десятичную цифру в младших 4-х битах регистра, а старшие 4 бита заполняет нулями. Используется команда в контексте:

*ADD AL, BL* ; сложить неупакованные цифры, находящиеся в  
*AAA* ; регистрах AL и BL и преобразовать результат в  
; правильное десятичное число.

Если результат операции превышает 9, то команда *AAA* добавляет 1 к содержимому регистра AL и устанавливает в 1 флаг CF, в противном случае флаг устанавливается в 0. Поскольку для дальнейшего анализа имеет значение только состояние флага CF, остальные флаги нужно считать

неопределенными (иначе говоря, их значения после команды AAA нельзя использовать для анализа – команд условного перехода).

Команда DAA преобразует содержимое регистра AL в 2 правильные упакованные десятичные цифры. Она используется в следующем контексте:

ADD AL, BL ; сложить упакованные BCD-числа в регистрах  
DAA ; AL и BL и преобразовать результат в упакованное  
; число.

Если результат превышает предельное значение для упакованных BCD-чисел, то DAA добавляет 1 к содержимому регистра AL и устанавливает в 1 флаг CF, в противном случае флаг устанавливается в 0. Замечания относительно остальных флагов для команды DAA такие же, как и для команды AAA.

Поскольку микропроцессор не имеет устройства вычитания, а имеет только устройство сложения (сумматор), вычитание на таком устройстве осуществляется в 2 этапа:

- а) меняется знак у вычитаемого - второго операнда или источника (иначе говоря, вычитаемое обращается);
- б) складываются уменьшаемое и обращенное вычитаемое.

Для обращения операнда в системе команд имеется самостоятельная команда *NEG приемник*. Эта команда вычитает значение операнда-приемника из нуля и, тем самым, формирует дополнительный код операнда (не забывайте, что дополнительный код числа в дополнительном коде будет являться модулем числа, или, иначе, обращение отрицательного числа даст число положительное). Установка флагов в этой команде осуществляется также, как в команде сложения.

Команды вычитания SUB и SBB аналогичны соответствующим командам сложения, только при вычитании флаг CF понимается как признак заема:

SUB приемник, источник– (приемник = приемник - источник)

SBB приемник, источник – (приемник = приемник - источник – перенос\_заем).

Ограничение при вычитании – нельзя вычесть значение регистра или ячейки памяти из константы, поскольку, например, команда *SUB 100, AL* недопустима. Однако, если заменить недопустимую операцию двумя следующими:

NEG AL

ADD AL, 100,

то вычитание из непосредственного значения будет выполнено и результат получен в AL.

Аналогично сложению, корректируются результаты вычитания при операциях с BCD-числами. Операция AAS корректирует результат вычитания неупакованной десятичной цифры из другой неупакованной десятичной цифры. Команда не имеет операндов и работает с регистром AL по следующему алгоритму:

а) если значение в регистре меньше или равно 9, то флаг CF устанавливается в 0 и управление передается следующей команде;

б) если значение в регистре AL больше 9,

- из содержимого младшей тетрады этого регистра вычитается 6,

- обнуляется старшая тетрада регистра AL;

- флаг CF устанавливается в 1, тем самым фиксируя наличие заема из предыдущего воображаемого разряда.

Проблему в операциях с десятичными числами создают ситуации, когда уменьшаемое меньше вычитаемого. В этом случае результат является отрицательным, однако, для десятичных чисел нет представления отрицательных значений. Выявление и обработка таких ситуаций полностью ложится на плечи программиста, что свидетельствует о том, что алгоритмы арифметических действий в ассемблере нужно разрабатывать более тщательно и подробнее, чем в языках программирования высокого уровня.



Для упакованных BCD-чисел коррекцию результата вычитания производят командой DAS. Как и в предыдущей команде коррекции, результат предполагается в регистре AL, но теперь в обеих тетрадах.

### 2.3. Умножение и деление целых чисел

В отличие от сложения и вычитания в микропроцессоре существуют по 2 операции умножения и деления: отдельно для чисел без знака и для чисел со знаком. Во всех командах присутствует только один операнд - источник. 2-ой операнд располагается в РОН. Для операции умножения правило операндов описано таблицей 4 [1].

Таблица 4.

Местоположение операндов и результата для умножения

Сомножитель_1	Сомножитель_2	Результат
Байт (в РОН или ячейке памяти)	AL	16 бит в AX: в AL – младшая часть результата, в AH – старшая
Слово (в РОН или ячейке памяти)	AX	32 бита в паре DX: AX; в AX – младшая часть результата, в DX – старшая часть результата

После выполнения команд флаги CF и OF показывают, какая часть произведения существенна для дальнейших операций. При умножении чисел без знака эти флаги равны 0, если старшая часть результата нулевая, в противном случае (результат превысил по значащим цифрам сомножители) флаги устанавливаются в 1. При умножении чисел со знаком флаги равны 0, если старшая половина произведения содержит расширение знакового разряда младшей половины (при положительном результате это 0, при отрицательном –1). **Обратите внимание, что эти операции не позволяют иметь в качестве операнда константу (непосредственное значение).**

Результат перемножения правильных неупакованных BCD-чисел может быть представлен в неупакованном формате BCD-чисел с помощью команды AAM. Команда работает с регистрами AL и AH и выполняет следующее: делит значение регистра AL на 10 и запоминает частное в регистре AH (старшая неупакованная цифра результата) и остаток – регистре

AL (младшая неупакованная цифра результата). Однако очевидно, что этими операциями можно выполнить только табличное умножение. Для более сложных операций умножения необходимо разработать программу, реализующую умножение “в столбик”, получение частных произведений, их сдвиги и сложение. Операции, аналогичной ААМ для упакованных BCD-чисел в микропроцессоре не существует.

В операции деления так же, как и при умножении учитывается знак, а правило расположения операндов отражается в таблице 5 [1].

Таблица 5

Местоположение операндов и результата для деления

Делимое	Делитель	Частное	Остаток
Слово 16 бит в регистре AX	Байт (в РОН или ячейке памяти)	Байт в регистре AL	Байт в регистре AH
32 бита в DX – старшая часть, в AX- младшая	Слово 16 бит (в РОН или ячейке памяти)	Слово 16 бит в регистре AX	Слово 16 бит в регистре DX

При делении может возникать прерывание ”деление на 0”. Такой результат может получиться не только, когда делитель равен 0, но и в следующих случаях:

- 1) при умножении чисел без знака для ситуации первой строки таблицы делимое более чем в 256 раз больше делителя,
- 2) при умножении чисел без знака для ситуации второй строки таблицы делимое более чем в 65636 раз больше делителя,
- 3) при делении чисел со знаком для ситуации первой строки таблицы делимое более чем в 128 раз больше значения делителя,
- 4) при делении чисел со знаком для ситуации второй строки таблицы делимое более чем в 32768 раз больше значения делителя.

Процесс выполнения деления двух неупакованных BCD-чисел может быть представлен в формате неупакованных BCD-чисел. Для этого **перед** операцией деления в регистре AX получают две неупакованные цифры делимого (выполняет эту операцию программист удобным для него

способом). Далее командой AAD преобразуется число в двоичное, которое затем является делимым в операции DIV. Причем в дальнейшей операции DIV двоичное число делится на неупакованную BCD-цифру, находящуюся в байтовом регистре или в байтовой ячейке памяти. Результат операции получается так, как описано первой строкой таблицы. Понятно, что с применением этих команд можно выполнять очень простые операции деления, но команду AAD можно использовать и в контексте преобразования упакованного (или неупакованного) десятичного числа из диапазона 00-99 в двоичный эквивалент.

К группе арифметических команд относят команды расширения операнда, которые называются командами преобразования:

CBW – преобразует байт в регистре AL в слово в регистре AX путем распространения старшего бита AL на все биты регистра AX;

CWD – преобразует слово в регистре AX в двойное слово в регистрах AX и DX, путем распространения старшего 15-ого бита регистра AX на все биты регистра DX.

Эти команды позволяют приводить разноформатные операнды к одному формату (большему).

### **3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

1. Изучить приведенный теоретический материал к лабораторной работе.
2. В соответствии с вариантом задания написать программу на ассемблере.
3. Оттранслировать программу в объектный код.
4. Провести отладку программы и проверить получаемые результаты.

### **4. СОДЕРЖАНИЕ ОТЧЕТА**

Отчет должен включать:

- титульный лист;
- описание цели работы;
- описание задания на лабораторную работу;

словесные пояснения к алгоритму решения задачи и схему программы (обращаю Ваше внимание на то, что понятие массива и индексации массива весьма условны, ибо в памяти ЭВМ элементы массива располагаются последовательно, строка за строкой. Поэтому физическая структура двумерного массива и вектора (одномерного массива) оказываются одинаковыми. Отличие двумерного массива и вектора заключается в интерпретации области памяти, отведенной этим структурам. Наращивание индекса элемента структуры определяется алгоритмом обработки структуры данных);

- листинги программ;
- результаты выполнения программ;
- выводы.

## **5. ВАРИАНТЫ ЗАДАНИЙ.**

Написать и отладить программы.

1. Подсчитать количество четных (нечетных) элементов двумерного массива
  - а) элементы массива однобайтовые;
  - б) элементы массива двухбайтовые.
2. Подсчитать сумму элементов строк (столбцов) двумерного массива
  - а) элементы массива однобайтовые;
  - б) элементы массива двухбайтовые.
3. Подсчитать сумму всех элементов двумерного массива и найти медиану (среднее арифметическое)
  - а) элементы массива однобайтовые;
  - б) элементы массива двухбайтовые.
4. Подсчитать суммы положительных и отрицательных элементов двумерного массива и определить, какая из них по абсолютной величине больше
  - а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

*Примечание к заданиям 1-2: в контексте определения четности-нечетности могут быть использованы различные команды ассемблера:*

*- команда логического умножения (конъюнкции). При этом учитывается то обстоятельство, что четные положительные числа имеют значение младшего двоичного разряда равное 0, а нечетные – 1. То есть, если результат выполнения команд*

*mov            al, mas[si]*

*and            al,1b*

*равен 0, то число четное, иначе – нечетное;*

*- команда арифметического сдвига вправо – shr на одну позицию. При этом число уменьшается в 2 раза, а младший разряд числа выталкивается во флаг переноса CF. Далее значение флага анализируется командой JC (переход, если есть перенос) или JNC (переход, если нет переноса);*

*- команда деления DIV или IDIV, например:*

*mov            al, mas[si] ;если исходные элементы однобайтовые*

*cbw                        ;преобразуем байт в полное слово*

*;если исходные элементы – двухбайтовые слова, то вместо предыдущих команд записываем команду mov        ax, mas[si]*

*div    byte ptr c        ;константа с объявлена в сегменте данных как c*  
*equ    2.*

*Следует отметить, что такая реализация обнаружения четности-нечетности самая неудачная, потому что операция деления относится к “длинным” операциям и занимает значительно больше процессорного времени, чем предыдущие команды. Кроме того, необходимо учитывать наличие или отсутствие знака у элементов массива. Приведенный пример предполагает, что элементы массива – беззнаковые.*

*Для отрицательных целых чисел признаки четности-нечетности будут иными!*

## **6. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Как представляются десятичные числа в ассемблере?
2. Когда может возникнуть ситуация переполнения при сложении или вычитании двоичных чисел?
3. Какие команды умножения и деления невозможны в ассемблере?
4. Какие команды используются при сложении (вычитании) десятичных чисел?
5. Какие команды используются при умножении (делении) десятичных чисел?
6. Сформулируйте правило операндов и результата для операции умножения (деления).
7. Когда возможна ситуация “деление на 0”?

## **Лабораторная работа № 5**

### **Программирование операций ввода-вывода**

#### **1. ЦЕЛЬ РАБОТЫ**

Целью работы является закрепление лекционного материала по командам прерывания на языке ассемблера и приобретение практических навыков использования этих команд в программах с операциями ввода-вывода.

#### **2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

**Прерывание** – это приостанов выполнения программы с целью выполнения какой-то более важной или нужной в данный момент другой программы или процедуры, после завершения которой продолжается выполнение программы с того места, где она была прервана. Прерывание позволяет компьютеру приостановить любое свое действие и временно переключиться на другое, как заранее запланированное, так и неожиданное, вызванное непредсказуемой ситуацией в работе машины или ее компонента. Каждое прерывание вызывает загрузку определенной программы, предназначенной для обработки возникшей ситуации – программы обработки прерываний.

Команды прерывания позволяют воспользоваться встроенными системными ресурсами (программами обработки прерываний) из программы пользователя. Под системными ресурсами понимаются программы, входящие в главную исполнительную программу ЭВМ, которая называется BIOS – базовая система ввода-вывода. В функции этой системы входит: запоминание символов, набираемых на клавиатуре, изображение символов на экране дисплея, обмен данными между устройствами, присоединенными к ЭВМ: дисководами, принтером и т.п. Чтобы обратиться к этим возможностям ЭВМ, пользовательская программа должна быть прервана и должна быть выполнена системная функция, после чего пользовательская программа продолжается со следующей после обращения к системным функциям

команды. Эти функциональные возможности и выполняют команды прерывания, которые описаны в таблице 6 [5].

Таблица 6

Формат команд прерывания

Мнемокод	Формат
<b>INT</b>	<b>INT тип_прерывания</b>
<b>INTO</b>	<b>INTO</b>
<b>IRET</b>	<b>IRET</b>

В команде прерывания INT тип\_прерывания представляет собой номер прерывания, которых в ЭВМ IBM насчитывается 256 (типы прерываний имеют номера 0-255):

- тип 0 – возникает при делении на 0 или если частное от деления превышает разрядную сетку;
- тип 1 – действует в режиме “трассировки” ( после выполнения каждой команды программы происходит останов);
- тип 2 – немаскируемые технические прерывания;
- тип 3 - прерывания по команде INT, включенной в программу (вызывает останов и отображение содержимого регистров микропроцессора);
- тип 4 – прерывание по команде INTO, включенной в программу (выполняется при условии, что при выполнении предыдущей команды произошло переполнение разрядной сетки);
- типы 8-15 – аппаратные прерывания, инициируемые внешними устройствами;
- типы 16- 31 – планируемые программные прерывания BIOS;
- типы 32-255 – программные прерывания DOS.

В некоторых типах прерываний BIOS и DOS имеется много разновидностей, иногда более 10. Так, прерывание 33 (21H) имеет около 100 разновидностей (это прерывание наиболее часто используется в программах пользователя). В таких случаях вид прерывания (внутри типа) определяется содержимым регистра АН.

Каждому прерыванию в памяти ЭВМ соответствует вектор прерывания (эти вектора размещены в оперативной памяти, начиная с нулевого адреса).



Каждый вектор прерывания размещается в 32-битовой ячейке памяти и представляет собой адрес, по которому размещена собственно программа прерывания. По сути такие программы очень похожи на процедуры, отличие в том, что программа прерывания заканчивается командой возврата IRET.

При выполнении команды INT микропроцессор производит следующие действия:

- а) помещает в стек регистр флагов;
- б) обнуляет флаг трассировки TF и флаг включения-выключения прерываний IF для блокировки других действий, кроме обработки вызванного прерывания;
- в) помещает в стек значение регистра CS;
- г) вычисляет адрес вектора прерываний, умножая номер\_прерывания на 4 (т.к. вектор прерывания занимает 4 байта или 32 бита);
- д) обращается ко второму слову из вычисленного адреса вектора прерываний и помещает его в регистр CS;
- е) помещает в стек значение указателя команд IP;
- ж) загружает в IP первое слово вектора прерываний.

После выполнения всех этих действий в стеке окажутся значения регистра флагов, адреса сегментного регистра CS и смещение команды, следующей за командой прерывания IP. Пара регистров CS:IP будет указывать на начальный адрес программы обработки прерывания, которую микропроцессор и начнет выполнять.

Команда INTO представляет собой команду условного прерывания. Она инициирует прерывание в том случае, когда флаг переполнения OF равен 1. Следовательно, применять эту команду надо после арифметических операций, которые могут вызвать переполнение. Однако обрабатываться прерывание будет только при наличии переполнения, при отсутствии эта команда будет игнорироваться. Команда INTO вызывает команду обработки по вектору прерывания 4, для определения выполняемых командой действий необходимо обратиться к техническому руководству для конкретной ЭВМ.

Выбор действий по обработке ситуаций переполнения возлагается на пользователя, поэтому без дополнительного уточнения по руководству конкретного компьютера пользоваться этой командой не рекомендую.

Команда IRET, как было уже сказано, является командой возврата после прерывания. Она извлекает из стека значения регистров CS и IP и регистра флагов (считывает три ячейки стека), а затем микропроцессор по новому содержанию регистров команд продолжит выполнение программы пользователя.

Наиболее распространенным в программах пользователя является использование прерывания 21H, которое предназначено для вызова функций DOS. В нижеприведенной таблице 7 [5] описываются только некоторые из видов этого прерывания.

Таблица 7

Наиболее часто используемые виды прерывания 21H

Значение АН	Операция	Дополнительные входные регистры	Выходные регистры
1	Ожидание набора символа на клавиатуре с последующим изображением его на экране	Не используются	(AL) = символ
2	Вывод символа на экран	(DL)= ASCII-код символа	Не используются
6	Чтение символа с клавиатуры	(DL)=0FFH	(AL) = символ
A	Чтение клавиатурной строки в буфер	(DS:DX)= адрес буфера, первый байт = размер буфера	Второй байт буфера = число прочитанных символов
9	Изображение строки на экране дисплея	(DS:DX)= адрес строки, которая должна заканчиваться символом \$	Не используются

Приведем некоторые примеры использования команд прерывания.

В диалоговых программах пользователя нередко требуется дать ответ на приглашение к вводу или сделать выбор из меню, нажав букву или цифру. Предположим, что в программе требуется дать ответ *Д* или *Н* на вопрос о продолжении или прекращении программы. Ввод *Д* заставляет программу перейти к группе команд, помеченных как *YES*, а *Н* – к командам с меткой *NO*. При ошибочном нажатии какой либо другой клавиши программа возвращается на ввод символа до тех пор, пока не будет нажата либо *Д*, либо *Н*.

```
GET_KEY MOV AH,01H      ; считать символ
          INT 21H
          CMP AL,'Д'      ; считан Д?
          JE  YES         ; если да, то перейти к YES
          CMP AL,'Н'      ; считан Н?
          JE  NO          ; если да, то перейти к YES
          JNE GET_KEY     ; иначе возврат на чтение символа.
```

В приведенном фрагменте распознаются только прописные буквы Д и Н, если Вы хотите, чтобы распознавались и строчные буквы, добавьте в программу соответствующие команды!

Во многих приложениях, требуется, чтобы пользователь ввел строку с информацией, например, свое полное имя (ФИО). Для этой цели служит функция *А* прерывания 21H. Чтобы воспользоваться этой функцией, в программе пользователя требуется зарезервировать в сегменте данных место для вводимой строки (в таблице это называется буфер строки). Количество выделяемых байтов должно быть на 2 больше максимального размера вводимой строки. Причем первый байт буфера должен задавать эту самую максимальную длину (фактически он будет содержать значение количества выделенных байтов памяти минус 2). Например, чтобы предусмотреть в программе ввод пользовательской строки из 50 символов, в сегменте данных нужно описать:

STRING DB 50, 51 DUP (?) ; первый байт в области – константа 50, за ней следуют незаполненные 51 байт для сообщения.

Чтение строки выполняется командами:

LEA DX, STRING; указатель на буфер поместить в DS:DX

MOV AH,0AH ; вызвать функцию A

INT 21H ; прочитать строку.

Функция A помещает количество фактически введенных символов во второй байт буфера STRING и не изменяет указатель DS:DX. Т.е. после выполнения команды INT первый информационный символ введенной строки находится по адресу (DX)+2.

Вряд ли хорошая пользовательская программа может обойтись без сообщений из программы. Это могут быть либо приглашения к вводу информации, либо сообщения о ходе выполнения программы. Такой сервис в программах на Ассемблере представляет функция 9 для работы с дисплеем. Приведем пример фрагмента программы для выдачи приглашения 'Введите ФИО'. Для функции 9 необходимо, чтобы текст сообщения заканчивался символом \$, поэтому в сегменте данных опишем:

MESS1 DB 'Введите ФИО: \$'.

В командном сегментном коде предусмотрим команды вызова функции 9:

LEA DX, MESS1

MOV AH,09H

INT 21H.

Имейте ввиду, что в этом случае курсор устанавливается в позицию \$, т.е. в то место строки, где мы хотим видеть фамилию. Чтобы после выдачи приглашения курсор установился в начале следующей строки, необходимо в текст выдаваемого приглашения перед символом доллара ввести символы возврата каретки и перехода на следующую строку, как это демонстрируется ниже:

MESS1 DB 'Введите ФИО', 0DH,0AH,'\$'.

### **3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

1. Изучить приведенный теоретический материал к лабораторной работе.
2. В соответствии с вариантом задания написать программу на ассемблере, предусмотрев вывод результатов работы программы на экран дисплея.
3. Оттранслировать программу в объектный код.
4. Провести отладку программы и проверить получаемые результаты.

### **4. СОДЕРЖАНИЕ ОТЧЕТА**

Отчет должен включать:

- титульный лист;
- описание цели работы;
- описание задания на лабораторную работу;
- словесные пояснения к алгоритму решения задачи и схему программы;
- листинги программ;
- результаты выполнения программ;
- выводы.

### **5. ВАРИАНТЫ ЗАДАНИЙ**

1. Найти первый (последний) максимальный (минимальный) элемент вектора и указать его местоположение:
  - а) элементы вектора однобайтовые;
  - б) элементы вектора двухбайтовые.
2. Найти максимальный (минимальный) элемент вектора и подсчитать количество таких элементов
  - а) элементы вектора однобайтовые;
  - б) элементы вектора двухбайтовые.
3. Найти первый (последний) максимальный(минимальный) элемент двумерного массива и указать его местоположение:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

4. Найти максимальный (минимальный) элемент двумерного массива и подсчитать количество таких элементов:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

5. Найти сумму всех отрицательных элементов двумерного массива, а среди положительных найти максимальный и указать его местоположение:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

6. Найти сумму всех положительных элементов двумерного массива, а среди отрицательных найти минимальный и указать его местоположение:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

7. Найти минимальный и максимальный элементы двумерного массива и указать их местоположение:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

8. Найти сумму элементов строк двумерного массива и определить строку (указать номер строки) с минимальной суммой элементов:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

9. Найти сумму элементов столбцов двумерного массива и определить столбец (указать номер столбца) с максимальной суммой элементов:

а) элементы массива однобайтовые;

б) элементы массива двухбайтовые.

***Примечания к реализации вывода значений элементов и их индексов на экран дисплея:***

***- десятичная цифра отличается от символа этой цифры на 30h, в чем легко убедиться, изучив таблицу кодов ASCII (приложение 3).***

*Например, если искомый максимальный элемент записан в переменную max, то фрагмент программы для его вывода на экран дисплея может быть следующим:*

```
mov      dl, max
add      dl, 30h
int      21h
```

*- предыдущий фрагмент применим в случае, если максимальное значение элемента является однозначным. Если элементы массива двухзначные (лежат в диапазоне от 10 до 99), то сначала необходимо выделить отдельные цифры, а затем их последовательно их распечатать:*

```
mov      al, max
cbw      ;преобразуем байт в полное слово
div      byte ptr c ;константа c объявлена в сегменте данных как c
equ      10
mov      dl,al      ;старшую цифру помещаем в dl
add      dl, 30h     ; и выводим ее на экран
int      21h
mov      dl,ah      ;младшую цифру помещаем в dl
add      dl, 30h     ; и выводим ее на экран
int      21h
```

## **6. КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Что такое прерывание?
2. Какую информацию содержит таблица векторов прерывания?
3. Каков механизм обработки прерывания?
4. Поясните команды программы, осуществляющие ввод информации с экрана дисплея.
5. Поясните команды программы, осуществляющие вывод информации с экрана дисплея.
6. Какие типы прерываний Вы знаете?

## СПИСОК ЛИТЕРАТУРЫ

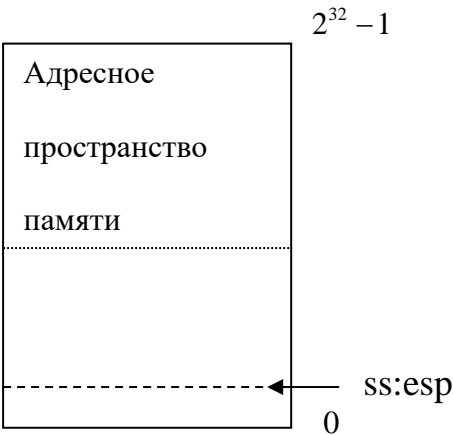
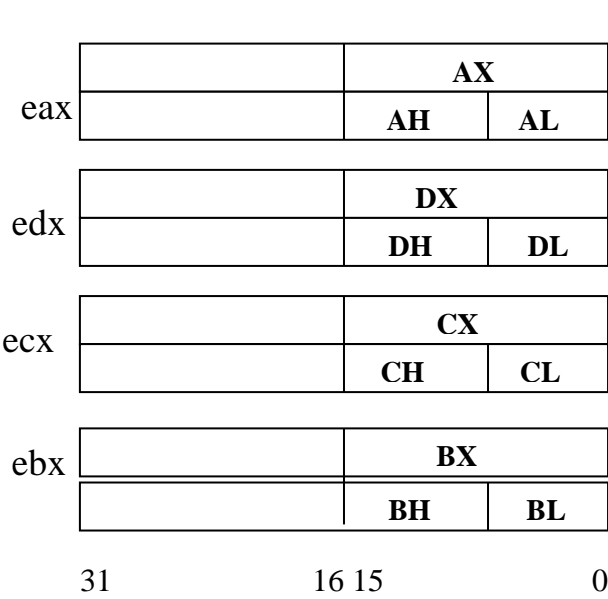
1. Assembler / В. Юров – СПб.: Питер, 2002 – 624 с.
2. Абель П. Язык ассемблера для IBM PC и программирования. М.: Высшая школа, 1992. – 447 с.
3. Бройдо В.Л. ПЭВМ: Архитектура и программирование на ассемблере. СПб.: СПб.ГИЭА, 1994. – 218 с.
4. Югов В.Ю., Хорошенко С. Assembler: учебный курс. СПб.: Питер, 1999. – 672 с.
5. Л. Скэнлон. Персональные ЭВМ IBM PC и XT. Программирование на языке Ассемблера: Пер. с англ. – 2-е изд., стереотип. – М.: Радио и связь. 1991. – 336 с.
6. Бройдо В.Л. Вычислительные системы, сети и телекоммуникации. – СПб.: Питер, 2002. – 688 с.
7. Ассемблер. Программирование простых алгоритмов обработки информации: Методические указания к лабораторным работам №№ 1, 2 / Сост. В. Н. Мукаеева, О.В. Даринцев; Уфимск. гос. авиац. техн. ун-т. – Уфа: УГАТУ, 2004 – 32 с.



Приложение 1

Программная модель микропроцессора Intel (Pentium III)

Регистры общего назначения  
целочисленного устройства



## Приложение 2

### Система команд микропроцессора Intel 8086

Мнемокод	Действие
1	2
AAA	Корректировка сложения для представления в кодах ASCII
FFD	Корректировка деления для представления в кодах ASCII
AAM	Корректировка умножения для представления в кодах ASCII
AAS	Корректировка вычитания для представления в кодах ASCII
ADC	Сложение с переносом
ADD	Сложение
AND	Логическое И
CALL	Вызов процедуры
CBW	Преобразование байта в слово
CLC	Обнуление флага переноса
CLI	Обнуление флага прерывания
CMC	Обращение флага переноса
CMP	Сравнение значений
CMPS, CMPSB, CMPSB	Сравнение строк
CWD	Преобразование слова в двойное слово
DAA	Корректировка сложения для представления в десятичной форме
DAS	Корректировка вычитания для представления в десятичной форме
DEC	Уменьшение значения на 1
DIV	Деление
ECS	Передача команды сопроцессору
HLT	Останов
IDIV	Деление целых чисел
IMUL	Умножение целых чисел
IN	Считывание значения из порта
INC	Приращение значения на 1
INT	Прерывание
INTO	Прерывание при переполнении

1	2
IRET	Возврат после прерывания
JA, JNBE	Переход, если выше
JAE, JNB	Переход, если выше или равно
JNC	Переход, если нет переноса
JB, JNAE	Переход, если ниже
JC	Переход, если есть перенос
JBE, JNA	Переход, если ниже или равно
JCXZ	Переход, если содержимое регистра CX равно 0
JE, JZ	Переход, если равно
JG, JNLE	Переход, если больше
JGE, JNL	Переход, если больше или равно
JL, JGNE	Переход, если меньше
JLE, JNG	Переход, если меньше или равно
JMP	Переход безусловный
JNE, JNZ	Переход, если не равно
JNO	Переход, если нет переполнения
JNP, JPO	Переход, если нет четности
JNS	Переход, если знаковый разряд = 0
JO	Переход, если переполнение
JP, JPE	Переход, если есть четность
JS	Переход, если знаковый разряд = 1
LAHF	Загрузка регистра AH флагами
LDS	Загрузка указателя с использованием регистра DS
LEA	Загрузка исполнительного адреса
LES	Загрузка указателя с использованием регистра ES
LOCK	Замыкание шины
LODS, LODSB, LODSW	Загрузка строки
LOOP	Повторение цикла до конца счетчика
LOOPE, LOOPZ	Повторение цикла, если равно
LOOPNE, LOOPNZ	Повторение цикла, если не равно
MOV	Пересылка значения
MOVS, MOVSB, MOVSW	Пересылка строки

1	2
MUL	Умножение
NEG	Обращение знака
NOP	Нет операции
NOT	Обращение битов
OR	Логическое ИЛИ
OUT	Вывод значения в порт
POP	Извлечение значения из стека
POPF	Извлечение флагов из стека
PUSH	Помещение значения в стек
PUSHF	Помещение флагов в стек
RCL	Сдвиг влево циклически с флагом переноса
RCR	Сдвиг вправо циклически с флагом переноса
REP, REPE, REPZ	Повторение, пока равно
REPNE, REPNZ	Повторение, пока не равно
RET	Возврат в вызывающий модуль (процедуру)
ROL	Сдвиг влево циклически
ROR	Сдвиг вправо циклически
SAHF	Загрузить флаги из регистра АН
SAL, SHL	Сдвиг влево арифметически
SAR	Сдвиг вправо арифметически
SBB	Вычитание с заемом
SCAS, SCASB, SCASW	Сканирование строки
SHR	Сдвиг вправо логически
STC	Установка флага переноса
STD	Установка флага направления
STI	Установка флага прерывания
STOS, STOSB, STOSW	Сохранение строки
SUB	Вычитание
TEST	Проверка
WAIT	Ожидание
XCHG	Обмен значений
XLAT	Выбор значения из таблицы
XOR	Логическое исключающее <b>ИЛИ</b>

Формат команд передачи управления

Мнемокод	Формат
Команды безусловной передачи управления	
CALL	CALL имя
RET	RET [число удаляемых из стека значений]
JMP	JMP имя
Команды условной передачи управления	
JA / JNBE	JA / JNBE близкая метка
JAЕ / JNB	JAЕ / JNB близкая метка
JNC	JNC близкая метка
JB / JNAE	JB / JNAE близкая метка
JC	JC близкая метка
JBE / JNA	JBE / JNA близкая метка
JCXZ	JCXZ близкая метка
JE / JZ	JE / JZ близкая метка
JG / JNLE	JG / JNLE близкая метка
JGE / JNL	JGE / JNL близкая метка
JL / JGNE	JL / JGNE близкая метка
JLE / JNG	JLE / JNG близкая метка
JNE / JNZ	JNE / JNZ близкая метка
JNO	JNO близкая метка
JNP / JPO	JNP / JPO близкая метка
JNS	JNS близкая метка
JO	JO близкая метка
JP / JPE	JP / JPE близкая метка
JS	JS близкая метка
Команды управления циклами	
LOOP	LOOP близкая метка
LOOPE / LOOPZ	LOOPE / LOOPZ близкая метка
LOOPNE / LOOPNZ	LOOPNE / LOOPNZ близкая метка

Формат арифметических команд

Мнемокод	Формат
Команды сложения ADD	ADD приемник, источник
ADC	ADC приемник, источник
AAA	AAA
DAA	DAA
INC	INC приемник
Команды вычитания SUB	SUB приемник, источник
SBB	SBB приемник, источник
AAS	AAS
DAS	DAS
DEC	DEC приемник
NEG	NEG приемник
CMP	CMP приемник, источник
Команды умножения MUL	MUL источник
IMUL	IMUL источник
AAM	AAM
Команды деления DIV	DIV источник
IDIV	IDIV источник
AAD	AAD
Команды расширения знака CBW	CBW
CWD	CWD











# Приложение 5

## Коды ASCII (диапазон 0-127)

Код	Симво	Код	Симво	Код	Символ	Код	Символ
0	NUL	16	DEL	32		48	0
1	SOH	17	DC1	33	!	49	1
2	STX	18	DC2	34	"	50	2
3	ETX	19	DC3	35	#	51	3
4	EDT	20	DC4	36	\$	52	4
5	ENQ	21	NAK	37	%	53	5
6	ACK	22	SYN	38	&	54	6
7	BEL	23	ETV	39	'	55	7
8	BS	24	CAN	40	(	56	8
9	HT	25	EM	41	)	57	9
10	LF	26	EOF	42	*	58	:
11	VT	27	ESC	43	+	59	;
12	FF	28	FS	44	,	60	<
13	CR	29	GS	45	-	61	=
14	SO	30	RS	46	.	62	>
15	SI	31	US	47	/	63	?
64	@	80	P	96	'	112	p
65	A	81	Q	97	a	113	q
66	B	82	R	98	b	114	r
67	C	83	S	99	c	115	s
68	D	84	T	100	d	116	t
69	E	85	U	101	e	117	u
70	F	86	V	102	f	118	v
71	G	87	W	103	g	119	w
72	H	88	X	104	h	120	x
73	I	89	Y	105	i	121	y
74	J	90	Z	106	j	122	z
75	K	91	[	107	k	123	{
76	L	92	\	108	l	124	
77	M	93	]	109	m	125	}
78	N	94	^	110	n	126	~
79	O	95	_	111	o	127	

Продолжение приложения 4

Альтернативная кодировка ГОСТа (диапазон 128-255)

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	А	144	Р	160	а	176	
129	Б	145	С	161	б	177	
130	В	146	Т	162	в	178	
131	Г	147	У	163	г	179	
132	Д	148	Ф	164	д	180	
133	Е	149	Х	165	е	181	
134	Ж	150	Ц	166	ж	182	
135	З	151	Ч	167	з	183	
136	И	152	Ш	168	и	184	
137	Й	153	Щ	169	й	185	
138	К	154	Ъ	170	к	186	
139	Л	155	Ы	171	л	187	
140	М	156	Ь	172	м	188	
141	Н	157	Э	173	н	189	
142	О	158	Ю	174	о	190	
143	П	159	Я	175	п	191	┘
192		208		224	р	240	Ë
193		209		225	с	241	ë
194		210		226	т	242	`
195	└	211		227	у	243	'
196	—	212		228	ф	244	'
197		213		229	х	245	`
198		214		230	ц	246	→
199		215		231	ч	247	←
200		216		232	ш	248	↓
201		217		233	щ	249	↑
202		218		234	ъ	250	÷
203		219		235	ы	251	±
204		220		236	ь	252	N'
205	=	221		237	э	253	□
206		222		238	ю	254	
207		223		239	я	255	