Методические указания по
выполнению лабораторных работ,
и организации самостоятельной работы
студентов по дисциплине

«Архитектура вычислительных систем»

Для студентов направления подготовки
Программная инженерия
(квалификация (степень) "бакалавр")

МИНИСТЕРСТВО ОБРАЗОВАНИЯ
И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Факультет систем управления

Кафедра автоматизации обработки информации (АОИ)

Методические указания

по выполнению лабораторных работ, и организации самостоятельной работы студентов по дисциплине

«*Архитектура вычислительных систем*»

Для студентов направления подготовки
Программная инженерия
(квалификация (степень) "бакалавр")

Разработчик:
доцент каф. АОИ
_____ Ю.Б. Гриценко

«____» _____ 2016г.

Томск – 2016

Содержание

Аннотация

Целью дисциплины «Архитектура вычислительных систем» является формирование у студента профессиональных знаний по теоретическим основам построения архитектуры ЭВМ и систем, их структурной и функциональной организации, программному обеспечению, эффективности и перспективам развития.

Процесс изучения дисциплины направлен на формирование следующей компетенции — владением архитектурой электронных вычислительных машин и систем (ОПК-2).

В результате изучения дисциплины студент должен:

Знать:

- принципы построения архитектуры ЭВМ и систем.

Уметь:

- производить сравнительный анализ различных архитектур электронных вычислительных машин и систем.

Владеть:

- навыками работы в среде различных электронных машин и систем.

Лабораторная работа № 1 «Изучение структуры программы на ассемблере»

1.1 Цель работы

Целью данной работы является изучение структуры программы на ассемблере, использование различных директив сегментации и создание примитивных программ, типа «Hello Word».

1.2 Структура программы на ассемблере

Программа на ассемблере представляет собой совокупность блоков памяти, называемых сегментами памяти. Программа может состоять из одного или нескольких таких блоков-сегментов. Каждый сегмент содержит совокупность предложений языка, каждое из которых занимает отдельную строку кода программы.

Предложения ассемблера бывают четырех типов:

- *команды или инструкции*, представляющие собой символические аналоги машинных команд. В процессе трансляции инструкции ассемблера преобразуются в соответствующие команды системы команд микропроцессора;
- *макрокоманды* – оформляемые определенным образом предложения текста программы, замещаемые во время трансляции другими предложениями;
- *директивы*, являющиеся указанием транслятору ассемблера на выполнение некоторых действий. У директив нет аналогов в машинном представлении;
- *строки комментариев*, содержащие любые символы, в том числе и буквы русского алфавита. Комментарии игнорируются транслятором.

1.2.1 Синтаксис ассемблера

Предложения, составляющие программу, могут представлять собой синтаксическую конструкцию, соответствующую команде, макрокоманде, директиве или комментарию. Для того чтобы транслятор ассемблера мог распознать их, они должны формироваться по определенным синтаксическим правилам.

Формат предложений ассемблера[1]:

```
Оператор директивы [;текст комментария]
Оператор команды [;текст комментария]
Оператор макрокоманды [; текст комментария]
```

Формат директив:

```
[Имя]                                    директива
[операнд1…операндN][;комментарий]
```

Формат команд и макрокоманд

```
[Имя                    метки:]          КОП
[операнд1…операндN][;комментарий]
```

Здесь:

- *имя метки* – идентификатор, значением которого является адрес первого байта того предложения исходного текста программы, которое он обозначает;
- *имя* – идентификатор, отличающий данную директиву от других одноименных директив;
- *код операции (КОП) и директива* – это мнемонические обозначения соответствующей машинной команды, макрокоманды или директивы транслятора;
- *операнды* – части команды, макрокоманды или директивы ассемблера, обозначающие объекты, над которыми производятся действия. Операнды ассемблера описываются выражениями с числовыми и текстовыми константами,

[1] Квадратные скобки означают не обязательные параметры

метками и идентификаторами переменных с использованием знаков операций и некоторых зарезервированных слов.

Допустимыми символами при написании текста программ являются:

- все латинские буквы: **A–Z**, **a–z**. При этом заглавные и строчные буквы считаются эквивалентными;
- цифры от **0** до **9**;
- знаки **?**, **@**, **$**, **_**, **&**;
- разделители **, . [ ] ( ) < > { } + / * % ! ' " ?** \ = # ^.

Предложения ассемблера формируются из *лексем*, представляющих собой синтаксически неразделимые последовательности допустимых символов языка, имеющие смысл для транслятора.

Лексемами являются:

- *идентификаторы* – последовательности допустимых символов, использующиеся для обозначения таких объектов программы, как коды операций, имена переменных и названия меток. Правило записи идентификаторов заключается в следующем: идентификатор может состоять из одного или нескольких символов. В качестве символов можно использовать буквы латинского алфавита, цифры и некоторые специальные знаки – _, ?, $, @. Идентификатор не может начинаться символом цифры. Длина идентификатора может быть до 255 символов, хотя транслятор воспринимает лишь первые 32, а остальные – игнорирует. Регулировать длину возможных идентификаторов можно с использованием опции командной строки **mv**. Кроме этого, существует возможность указать транслятору на то, чтобы он различал прописные и строчные буквы либо игнорировал их различие (что и делается по умолчанию). Для этого применяются опции командной строки **/mu, /ml, /mx**;
- *цепочки символов* – последовательности символов, заключенные в одинарные или двойные кавычки;
- *целые числа* в одной из следующих систем счисления: *двоичной, десятичной, шестнадцатеричной*.

Отождествление чисел при записи их в программах на ассемблере производится по определенным правилам:

- **Десятичные числа** не требуют для своего отождествления указания каких-либо дополнительных символов, например 25 или 139.
- Для отождествления в исходном тексте программы **двоичных чисел** необходимо после записи нулей и единиц, входящих в их состав, поставить латинское “**b**”, например 10010101**b**.
- **Шестнадцатеричные числа** имеют больше условностей при своей записи:
  - *Во-первых*, они состоят из цифр **0...9**, строчных и прописных букв латинского алфавита **a**, **b**, **c**, **d**, **e**, **f** или **A**, **B**, **C**, **D**, **E**, **F**.
  - *Во-вторых*, у транслятора могут возникнуть трудности с распознаванием шестнадцатеричных чисел из-за того, что они могут состоять как из одних цифр 0...9 (например, 190845), так и начинаться с буквы латинского алфавита (например, **ef15**). Для того чтобы "объяснить" транслятору, что данная лексема не является десятичным числом или идентификатором, программист должен специальным образом выделять шестнадцатеричное число. Для этого, на конце последовательности шестнадцатеричных цифр, составляющих шестнадцатеричное число, записывают латинскую букву “**h**”. Это обязательное условие. Если шестнадцатеричное число начинается с буквы, то перед ним записывается ведущий ноль: **0**ef15**h**.

1.2.2 Директивы сегментации

В ходе предыдущего обсуждения мы выяснили все основные правила записи команд и операндов в программе на ассемблере. Открытым остался вопрос о том, как правильно

оформить последовательность команд, чтобы транслятор мог их обработать, а микропроцессор – выполнить.

При рассмотрении архитектуры микропроцессора мы узнали, что он имеет шесть сегментных регистров, посредством которых может одновременно работать:

- с одним сегментом кода;
- с одним сегментом стека;
- с одним сегментом данных;
- с тремя дополнительными сегментами данных.

Еще раз вспомним, что физически сегмент представляет собой область памяти, занятую командами и (или) данными, адреса которых вычисляются относительно значения в соответствующем сегментном регистре.

Синтаксическое описание сегмента на ассемблере представляет собой следующую конструкцию:

```
Имя сегмента SEGMENT [тип выравнивания] [тип комбинирования] [класс сегмента] [тип размера сегмента]
    …
    содержимое сегмента
    …
Имя сегмента ENDS
```

Важно отметить, что функциональное назначение сегмента несколько шире, чем простое разбиение программы на блоки кода, данных и стека. Сегментация является частью общего механизма, связанного с концепцией модульного программирования. Она предполагает унификацию оформления объектных модулей, создаваемых компилятором, в том числе с разных языков программирования. Это позволяет объединять программы, написанные на разных языках. Именно для реализации различных вариантов такого объединения и предназначены операнды в директиве SEGMENT. Рассмотрим их подробнее:

- *Атрибут выравнивания сегмента* (тип выравнивания) сообщает компоновщику о том, что нужно обеспечить

размещение начала сегмента на заданной границе. Это важно, поскольку при правильном выравнивании доступ к данным в процессорах i80x86 выполняется быстрее. Допустимые значения этого атрибута следующие:

- BYTE – выравнивание не выполняется. Сегмент может начинаться с любого адреса памяти;
- WORD – сегмент начинается по адресу, кратному двум, то есть последний (младший) значащий бит физического адреса равен 0 (выравнивание на границу слова);
- DWORD – сегмент начинается по адресу, кратному четырем, то есть два последних (младших) значащих бита, равны 0 (выравнивание на границу двойного слова);
- PARA – сегмент начинается по адресу, кратному 16, то есть последняя шестнадцатеричная цифра адреса должна быть 0h (выравнивание на границу параграфа);
- PAGE – сегмент начинается по адресу, кратному 256, то есть две последние шестнадцатеричные цифры должны быть 00h (выравнивание на границу 256-байтной страницы);
- MEMPAGE – сегмент начинается по адресу, кратному 4 Кбайт, то есть три последние шестнадцатеричные цифры должны быть 000h (адрес следующей 4-Кбайтной страницы памяти).

По умолчанию тип выравнивания имеет значение PARA.

- *Атрибут комбинирования сегментов* (комбинаторный тип) сообщает компоновщику, как нужно комбинировать сегменты различных модулей, имеющие одно и то же имя. Значениями атрибута комбинирования сегмента могут быть:
  - PRIVATE – сегмент не будет объединяться с другими сегментами с тем же именем вне данного модуля;
  - PUBLIC – заставляет компоновщик соединить все сегменты с одинаковыми именами. Новый объединенный сегмент будет целым и непрерывным. Все адреса (смещения) объектов, а это могут быть, в зависимости от типа сегмента, команды и данные, будут

вычисляться относительно начала этого нового сегмента;

- COMMON – располагает все сегменты с одним и тем же именем по одному адресу. Все сегменты с данным именем будут перекрываться и совместно использовать память. Размер полученного в результате сегмента будет равен размеру самого большого сегмента;
- AT xxxx – располагает сегмент по абсолютному адресу параграфа (параграф – объем памяти, кратный 16, поэтому последняя шестнадцатеричная цифра адреса параграфа равна 0);
- STACK – определение сегмента стека. Заставляет компоновщик соединить все одноименные сегменты и вычислять адреса в этих сегментах относительно регистра ss.

По умолчанию атрибут комбинирования принимает значение PRIVATE.

- *Атрибут класса сегмента* (тип класса) – это заключенная в кавычки строка, помогающая компоновщику определить соответствующий порядок следования сегментов при собирании программы из сегментов нескольких модулей. Компоновщик объединяет в памяти все сегменты с одним и тем же именем класса (имя класса, в общем случае, может быть любым, но лучше, если оно будет отражать функциональное назначение сегмента);
- *Атрибут размера сегмента*. Для процессоров i80386 и выше сегменты могут быть 16 или 32-разрядными. Это влияет, прежде всего, на размер сегмента и порядок формирования физического адреса внутри него. Атрибут может принимать следующие значения:
  - USE16 – это означает, что сегмент допускает 16-разрядную адресацию. При формировании физического адреса может использоваться только 16-разрядное смещение. Соответственно, такой сегмент может содержать до 64 Кбайт кода или данных;
  - USE32 – сегмент будет 32-разрядным. При формировании физического адреса может

использоваться 32-разрядное смещение. Поэтому такой сегмент может содержать до 4 Гбайт кода или данных.

Все сегменты сами по себе равноправны, так как директивы SEGMENT и ENDS не содержат информации о функциональном назначении сегментов. Для того чтобы использовать их как сегменты кода, данных или стека, необходимо предварительно сообщить транслятору об этом, для чего используют специальную директиву ASSUME. Эта директива сообщает транслятору о том, какой сегмент, к какому сегментному регистру привязан. В свою очередь, это позволит транслятору корректно связывать символические имена, определенные в сегментах.

Далее приведем пример программы с использованием стандартных директив сегментации:

```
data segment para public 'data' ;сегмент
данных
     message db 'Hello World,$' ; описание
строки
data ends

stk segment     stack
     db 256 dup ('?') ; размер сегмента стека
stk ends

code segment para public 'code' ; начало
сегмента
 ; кода
main proc ;начало процедуры main
     assume cs:code,ds:data,ss:stk
 mov  ax,data ; адрес сегмента данных
 ; в регистр ax
     mov  ds,ax ; ax в ds

     ...

     mov  ah,9
     mov  dx,offset message
```

```
     int  21h ; ah=9 функция 21h прерывания
 ; выводит строку на экран, адрес
 ; которой храниться в регистре dx,
 ; строка должна обязательно
 ; заканчиваться символом $
     ...

     mov  ax,4c00h ; пересылка 4c00h в
регистр ax
     int  21h ; вызов прерывания с номером
21h
main  endp ; конец процедуры main
code  ends ; конец сегмента кода
end   main ; конец программы с точкой входа
main
```

Для простых программ, содержащих по одному сегменту для кода, данных и стека, хотелось бы упростить ее описание. Для этого в трансляторы MASM и TASM ввели возможность использования *упрощенных директив сегментации*. Но здесь возникла проблема, связанная с тем, что необходимо было как-то компенсировать невозможность напрямую управлять размещением и комбинированием сегментов. Для этого, совместно с упрощенными директивами сегментации, стали использовать директиву указания модели памяти **MODEL**, которая частично стала управлять размещением сегментов и выполнять функции директивы **ASSUME** (поэтому при использовании упрощенных директив сегментации директиву ASSUME можно не использовать). Эта директива связывает сегменты, которые в случае использования упрощенных директив сегментации имеют предопределенные имена с сегментными регистрами (хотя явно инициализировать *ds* все равно придется).

Теперь, перепишем вышеприведенную программу с использованием упрощенных директив сегментации.

```
masm ;режим работы TASM: ideal или masm
model small ; модель памяти
```

```
.data ; сегмент данных
 message db 'Hello World,$' ; описание строки

.stack ;сегмент стека
 db 256 dup ('?') ; сегмент стека

.code ;сегмент кода
main proc ; начало процедуры main
 mov ax,@data ; заносим адрес сегмента данных
в
 ; регистр ax
 mov ds,ax ;ax в ds

      ...

 mov  ah,9
 mov  dx,offset message
 int  21h ; ah=9 функция 21h прерывания
 ; выводит строку на экран, адрес
 ; которой храниться в регистре dx
      ...

 mov ax,4c00h ; пересылка 4c00h в регистр ax
 int 21h ; вызов прерывания с номером 21h
main endp ; конец процедуры main
end main ; конец программы с точкой входа main
```

Обязательным параметром директивы MODEL является *модель памяти*. Этот параметр определяет модель сегментации памяти для программного модуля. Предполагается, что программный модуль может иметь только определенные типы сегментов, которые определяются упомянутыми нами ранее *упрощенными директивами описания сегментов*. Эти директивы приведены в таблице 1.

Таблица 1. Упрощенные директивы определения сегмента

Формат директивы (режим MASM)	**Формат директивы (режим IDEAL)**	**Назначение**
.CODE [имя]	CODESEG[имя]	Начало или продолжение сегмента кода
.DATA	DATASEG	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near
.CONST	CONST	Начало или продолжение сегмента постоянных данных (констант) модуля
.FARDATA [имя]	FARDATA [имя]	Начало или продолжение сегмента инициализированных данных типа far

Наличие в некоторых директивах параметра **[имя]** говорит о том, что возможно определение нескольких сегментов этого типа. С другой стороны, наличие нескольких видов сегментов данных обусловлено требованием обеспечения совместимости с некоторыми компиляторами языков высокого уровня, которые создают разные сегменты данных для инициализированных и неинициализированных данных, а также констант.

При использовании директивы **MODEL** транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы, с тем, чтобы получить информацию о тех или иных характеристиках данной модели памяти (таблица 3). Перечислим эти идентификаторы и их значения (табл. 2).

Таблица 2. Модели памяти

Модель	**Тип**	**Тип**	**Назначение модели**

	кода	**данных**	
TINY	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com.
SMALL	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP. Эту модель обычно используют для большинства программ на ассемблере
MEDIUM	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления – типа far. Данные объединены в одной группе; все ссылки на них – типа near
COMPACT	near	far	Код в одном сегменте; ссылка на данные – типа far
LARGE	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль

Параметр модификатор директивы MODEL позволяет уточнить некоторые особенности использования выбранной модели памяти (табл. 3).

Таблица 3. Модификаторы модели памяти

Значение модификатора	**Назначение**
use16	Сегменты выбранной модели используются как 16-битные (если соответствующей

	директивой указан процессор i80386 или i80486)
use32	Сегменты выбранной модели используются как 32-битные (если соответствующей директивой указан процессор i80386 или i80486)
dos	Программа будет работать в MS-DOS

Необязательные параметры – язык и модификатор языка, определяют некоторые особенности вызова процедур. Необходимость в использовании этих параметров появляется при написании и связывании программ на различных языках программирования.

Описанные нами стандартные и упрощенные директивы сегментации не исключают друг друга. Стандартные директивы используются, когда программист желает получить полный контроль над размещением сегментов в памяти и их комбинированием с сегментами других модулей.

Упрощенные директивы целесообразно использовать для простых программ и программ, предназначенных для связывания с программными модулями, написанными на языках высокого уровня. Это позволяет компоновщику эффективно связывать модули разных языков за счет стандартизации связей и управления.

1.2.3 Создание COM-программ

Все вышеприведенные директивы сегментации и примеры программ предназначены для создания программ в EXE-формате[2]. Компоновщик LINK автоматически генерирует особый формат для EXE-файлов, в котором присутствует специальный начальный блок (заголовок) размером не менее 512 байт.

[2] За исключением модели памяти TINY при использовании упрощенных директив сегментации.

Для выполнения можно также создавать COM-файлы. Примером часто используемого COM-файла является COMMAND.COM.

Размер программы. EXE-программа может иметь любой размер, в то время как COM-файл ограничен размером одного сегмента и не превышает 64К. COM-файл всегда меньше, чем соответствующий EXE-файл; одна из причин этого - отсутствие в COM-файле 512-байтового начального блока EXE-файла.

Сегмент стека. В EXE-программе определяется сегмент стека, в то время как COM-программа генерирует стек автоматически. Таким образом, при создании ассемблерной программы, которая будет преобразована в COM-файл, стек должен быть опущен.

Сегмент данных. В EXE программе обычно определяется сегмент данных, а регистр DS инициализируется адресом этого сегмента. В COM-программе все данные должны быть определены в сегменте кода. Ниже будет показан простой способ решения этого вопроса.

Инициализация. EXE-программа записывает нулевое слово в стек и инициализирует регистр DS. Так как COM-программа не имеет ни стека, ни сегмента данных, то эти шаги отсутствуют.

Когда COM-программа начинает работать, все сегментные регистры содержат адрес префикса программного сегмента (PSP), – 256-байтового (шест. 100) блока, который резервируется операционной системой DOS непосредственно перед COM или EXE программой в памяти. Так как адресация начинается с шест. смещения 100 от начала PSP, то в программе после оператора SEGMENT кодируется директива ORG 100H.

Обработка. Для программ в EXE и COM форматах выполняется ассемблирование для получения OBJ-файла, и компоновка для получения EXE-файла. Если программа создается для выполнения как EXE-файл, то ее уже можно выполнить. Если же программа создается для выполнения как COM-файл, то компоновщиком будет выдано сообщение:

Warning: No STACK Segment

(Предупреждение: Сегмент стека не определен)

Ниже приведем пример COM-программы:

```
CSEG  Segment 'Code'
assume CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
org   100h
start:
      ...

      mov   ah,9
      mov   dx,offset message
      int   21h ; ah=9 функция 21h прерывания
 ; выводит строку на экран, адрес
 ; которой храниться в регистре dx
      ...

 int 20h ; выход из COM-программы
 message db 'Hello World,$' ; описание строки
ends
end start
```

1.3 Компиляция программ на ассемблере

Для написания программ на языке ассемблере вы можете воспользоваться любым текстовым редактором, поддерживающим кодировку ASCII-символов, например «Блокнот»/«Notepad» из ОС Windos или встроенным текстовым редактором FAR/DN/NC и др.

Для создания исполняемых файлов из программ написанных на языке ассемблере вам необходимо использовать компилятор TASM.EXE и линковщик TLINK.EXE.

TASM.EXE компилирует программные модули ассемблера в объектные модули OBJ. А TLINK.EXE из нескольких модулей делает один исполняемый файл EXE или COM. Более подробный синтаксис использования TASM.EXE и TLINK.EXE можно получить запустив эти программы без параметров.

1.4 Задание на выполнение

1. Ознакомиться со структурой программы на ассемблере.
2. Написать и скомпилировать программу «Hello World» с использованием стандартных директив компиляции в exe-формате.
3. Написать и скомпилировать программу «Hello World» с использованием упрощенных директив компиляции в exe-формате.
4. Написать и скомпилировать программу «Hello World» с использованием стандартных директив компиляции в com-формате.
5. Написать и скомпилировать программу «Hello World» с использованием упрощенных директив компиляции в com-формате.
6. Подготовиться к сдаче лабораторной работы по теоретической части лабораторной работы.

ЛАБОРАТОРНАЯ РАБОТА №2 «Изучение функций ввода/вывода»

2.1. Цель работы

Целью данной работы является изучение функций ввода/вывода.

2.2 Функции прерываний ввода/вывода

В операционной системе существует большая группа функций 21h прерывания (прерывания DOS). Небольшую часть этих функций составляют функции ввода вывода информации.

Для вызова какого-либо прерывания необходимо:

- в регистр AH занести номер функции прерывания;
- в зависимости от типа прерывания в какие-либо регистры занести дополнительные параметры;
- использовать команду int с указанием номера прерывания.

С 9h функцией прерывания 21h вы познакомились на предыдущей лабораторной работе.

В данной лабораторной работе вам понадобится помимо функции вывода строки использовать функции ввода и вывода символа.

Для вызова функции ввода символа используют 1h-функцию 21h прерывания. После нажатия символа на клавиатуре в регистре AL сохраняется код ASCII нажатого символа.

Для вывода символа на экран можно воспользоваться функцией 2h прерывания 21h. В регистр AL помещается ASCII-код символа и вызывается прерывание 21h.

Дополнительную информацию по различным функциям прерываний операционной системы можно взять в электронном справочнике thelp (рис. 2.1-2.4).

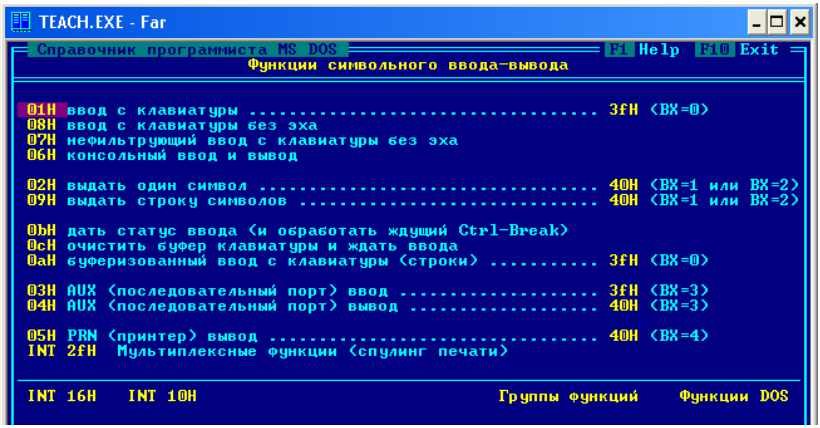


Рис. 2.1 – Функции символьного ввода/вывода

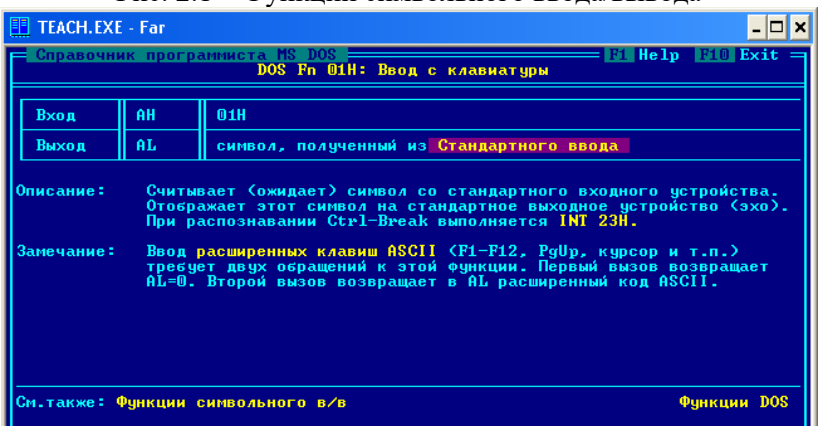


Рис. 2.2 – Функция ввод с клавиатуры

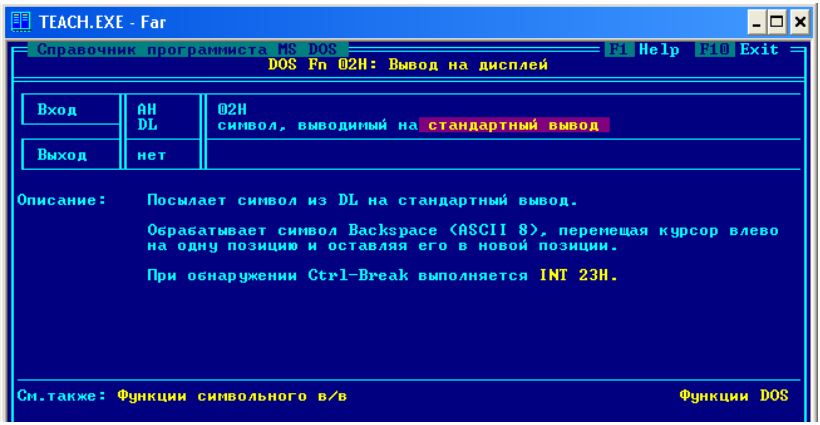


Рис. 2.3 Функция вывода символа на дисплей

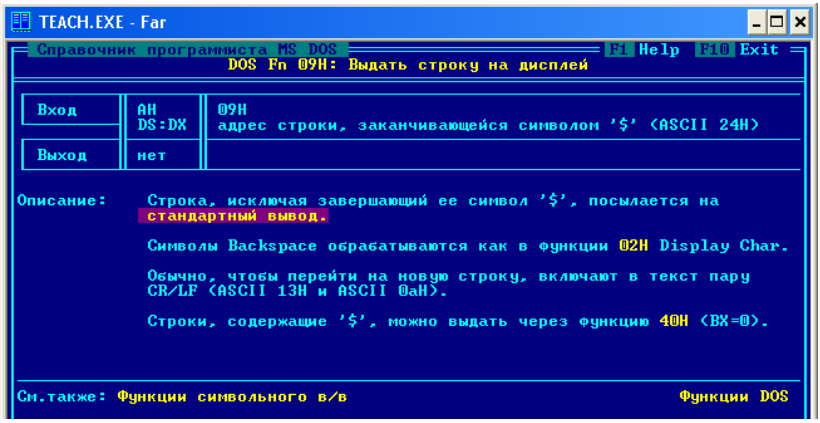


Рис. 2.4 – Функция вывода строки на дисплей

2.3 Примеры использования функций ввода/вывода

Пример программы ввода шестнадцатеричного числа:

```
data    segment para public 'data'    ;сегмент данных
message db      'Введите две шестнадцатеричные цифры,$'
data    ends
stk     segment stack
        db      256 dup ('?')  ;сегмент стека
stk     ends
```

```
code    segment para public 'code'    ;начало сегмента кода
main    proc    ;начало процедуры main
        assume cs:code,ds:data,ss:stk
        mov     ax,data ;адрес сегмента данных в регистр ax
        mov     ds,ax   ;ax в ds
        mov     ah,9
        mov     dx,offset message
        int     21h
xor     ax,ax   ;очистить регистр ax
        mov     ah,1h   ;1h в регистр ah
        int     21h     ;генерация прерывания с номером 21h
        mov     dl,al   ;содержимое регистра al в регистр dl
        sub     dl,30h  ;вычитание: (dl)=(dl)-30h
        cmp     dl,9h   ;сравнить (dl) с 9h
        jle     M1      ;перейти на метку M1 если dl<9h или dl=9h
        sub     dl,7h   ;вычитание: (dl)=(dl)-7h
M1:     mov     cl,4h   ;пересылка 4h в регистр cl
        shl     dl,cl   ;сдвиг содержимого dl на 4 разряда влево
        int     21h     ;вызов прерывания с номером 21h
        sub     al,30h  ;вычитание: (dl)=(dl)-30h
        cmp     al,9h   ;сравнить (al) с 9h   28
        jle     M2      ;перейти на метку M2 если al<9h или al=9h
        sub     al,7h   ;вычитание: (al)=(al)-7h
M2:     add     dl,al   ;сложение: (dl)=(dl)+(al)
mov     ax,4c00h        ;пересылка 4c00h в регистр ax
        int     21h     ;вызов прерывания с номером 21h
main    endp            ;конец процедуры main
code    ends            ;конец сегмента кода
end     main            ;конец программы с точкой входа main
```

Пример кода программы вывода шестнадцатеричного двухзначного числа:

```
        mov bl,16       ; делитель
        xor ax,ax       ; обнуляем ax
        mov al,rez      ;  заносим в al число на которое нужно
напечатать
        div bl          ;  ax делим на bl, в al частное, ah
остаток
        push ax         ; сохраняем ax в стек
        mov dl,al       ; готовим к печати первую цифру числа
        add dl,30h
        mov ah,02       ; номер функции вывода символа
        int 21h         ; печатаем первую цифру числа
        pop ax          ; достаем ax из стека
        mov dl,ah       ; готовим к печати вторую цифру числа
        add dl,30h
        mov ah,02       ; номер функции вывода символа
        int 21h         ; печатаем вторую цифру числа
```

Пример кода программы вывода двоичного числа:

```
mov     cx,16
```

```
m1:                    ; число которое необходимо вывести на
экран
       sal bx,1              ; арифм. сдвиг влево на 1 бит,
значение бита
                       ; попадает в флаг cf
       adc dl,30h      ; dl=dl+30h+cf
       xor dx,dx       ; обнуляем dx (dl – младшая часть dx)
       mov ah,02       ; номер функции вывода символа
       int 21h
       loop m1         ; если cx<>0 то cx=cx-1 и переход на
метку m1
```

2.4 Задание на выполнение

Написать программу ввода двух шестнадцатеричных чисел и вывода на экран этих чисел в двоичном виде.

ЛАБОРАТОРНАЯ РАБОТА №3 «Изучение арифметических и логических команд»

3.1. Цель работы

Целью данной работы является изучение арифметических и логических команд микропроцессора.

3.2 Арифметические команды

Сложение двоичных чисел без знака. Микропроцессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда. Например, при сложении операндов размером в байт результат не должен превышать число 255. Если это происходит, то результат оказывается неверным. К примеру, выполним сложение: 254 + 5 = 259 в двоичном виде. 11111110 + 0000101 = 1 00000011. Результат вышел за пределы восьми бит и правильное его значение укладывается в 9 бит, а в 8-битовом поле операнда осталось значение 3, что, конечно, неверно. В микропроцессоре этот исход сложения прогнозируется и предусмотрены специальные средства для фиксирования подобных ситуаций и их обработки. Так, для фиксирования ситуации выхода за разрядную сетку результата, как в данном случае, предназначен флаг переноса cf. Он располагается в бите 0 регистра флагов eflags/flags. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для корректировки. Это предполагает включение участков кода после операции сложения, в которых анализируется флаг cf. Анализ этого флага можно провести различными способами. Самый простой и доступный – использовать команду условного перехода jcc. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода.

Переход на эту метку осуществляется в случае, если в результате работы предыдущей команды флаг cf установился в 1. В системе команд микропроцессора имеются три команды двоичного сложения:

- inc операнд – операция инкремента, то есть увеличения значения операнда на 1;
- add операнд_1,операнд_2 – команда сложения с принципом действия: операнд_1 = операнд_1 + операнд_2
- adc операнд_1,операнд_2 – команда сложения с учетом флага переноса cf: операнд_1 = операнд_1 + операнд_2 + значение_cf

Обратите внимание на последнюю команду – это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда adc является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Рассмотрим пример вычисления суммы чисел:

```
<1> ;prg1
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> a db 254
<7> .code ;сегмент кода
<8> main:
<9> mov ax,@data
<10> mov ds,ax
<11> ...
<12> xor ax,ax
<13> add al,17
<14> add al,a
<15> jnc m1;если нет переноса, то перейти
 ;на m1
<16> adc ah,0;в ax сумма с учетом переноса
<17> m1: ...
<18> exit:
<19> mov ax,4c00h ;стандартный выход
<20> int 21h
<21> end main ;конец программы
```

В строках 13–14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывается строкой 15, где команда jnc (хотя можно было обойтись и без нее) проверяет состояние флага cf. Если он установлен в 1, то это признак того, что результат операции получился больше по размеру, чем размер операнда, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера AX, для чего учитываем перенос в старший разряд командой ADC (строка 15).

Сложение двоичных чисел со знаком. Микропроцессор не подозревает о различии между числами со знаком и без знака. Вместо этого у него есть средства фиксирования возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели при обсуждении сложения чисел без знака:

- флаг переноса cf, установка которого в 1 говорит о том, что произошел выход за пределы разрядности операндов;
- команду adc, которая учитывает возможность такого выхода (перенос из младшего разряда).

Другое средство – это регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения of в регистре eflags (бит 11).

Дополнительно к флагу of при переносе из старшего разряда устанавливается в 1 и флаг переноса cf. Так как микропроцессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Проанализировать флаги cf и of можно командами условного перехода jc\jnc и jo\jno соответственно.

Что же касается команд сложения чисел со знаком, то они те же, что и для чисел без знака.

Вычитание двоичных чисел без знака. Как и при анализе операции сложения, порассуждаем над сутью процессов, происходящих при выполнении операции вычитания. Если

уменьшаемое больше вычитаемого, то проблем нет, – разность положительна, результат верен. Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо завернуть. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Микропроцессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим, в разрядной сетке операнда.

Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага cf. Если он установлен в 1, то это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения, группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом. К командам вычитания относятся:

- dec операнд – операция декремента, то есть уменьшения значения операнда на 1;
- sub операнд_1,операнд_2 – команда вычитания; ее принцип действия:
  операнд_1 = операнд_1 – операнд_2
- sbb операнд_1,операнд_2 – команда вычитания с учетом заема (флага cf): операнд_1 = операнд_1 – операнд_2 – значение_cf

Как видите, среди команд вычитания есть команда sbb, учитывающая флаг переноса cf. Эта команда подобна adc, но теперь уже флаг cf выполняет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример программной обработки ситуации:

```
<1> ;prg2
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> .code ;сегмент кода
<7> main: ;точка входа в программу
<8> ...
```

```
<9> xor ax,ax
<10> mov al,5
<11> sub al,10
<12> jnc m1 ;нет переноса?
<13> neg al ;в al модуль результата
<14> m1: ...
<15> exit:
<16> mov ax,4c00h ;стандартный выход
<17> int 21h
<18> end main ;конец программы
```

В этом примере в строке 11 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному виду (получить его модуль), применяется команда neg, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение дополнения или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага cf. Дальше все зависит от алгоритма обработки.

Вычитание двоичных чисел со знаком. Здесь все несколько сложнее. Последний пример показал то, что микропроцессору незачем иметь два устройства – сложения и вычитания. Достаточно наличия только одного – устройства сложения. Но для вычитания способом сложения чисел со знаком в дополнительном коде необходимо представлять оба операнда – и уменьшаемое, и вычитаемое. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего, они связаны с тем, что старший бит операнда рассматривается как знаковый.

Отследить ситуацию переполнения мантиссы можно по содержимому флага переполнения of. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера, и программист должен предусмотреть действия по корректировке результата.

Умножение чисел без знака. Для умножения чисел без знака предназначена команда

mul сомножитель_1

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд – сомножитель_2 задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и размещения второго операнда и результата приведены в таблице 4.

Таблица 4. Расположение операндов и результата при умножении

сомножитель_1	**сомножитель_2**	**Результат**
Байт	al	16 бит в ax: al – младшая часть результата; ah – старшая часть результата
Слово	ax	32 бит в паре dx:ax: ax – младшая часть результата; dx – старшая часть результата
Двойное слово	eax	64 бит в паре edx:eax: eax – младшая часть результата; edx – старшая часть результата

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах – на месте сомножитель_2 (младшая часть) и в

дополнительном регистре ah, dx, edx (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра, и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам по предыдущему обсуждению флаги переноса cf и переполнения of:

- если старшая часть результата нулевая, то после операции произведения флаги cf = 0 и of = 0;
- если же эти флаги ненулевые, то это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Умножение чисел со знаком. Для умножения чисел со знаком предназначена команда

imul операнд_1[,операнд_2,операнд_3]

Эта команда выполняется так же, как и команда mul. Отличительной особенностью команды imul является только формирование знака.

Если результат мал и умещается в одном регистре (то есть если cf = of = 0), то содержимое другого регистра (старшей части) является расширением знака – все его биты равны старшему биту (знаковому разряду) младшей части результата.

В противном случае, (если cf = of = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата.

Если вы посмотрите описание команды imul, то увидите, что она допускает более широкие возможности по заданию местоположения операндов. Это сделано для удобства использования.

Деление чисел без знака. Для деления чисел без знака предназначена команда:

div делитель

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бит. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка.

Варианты местоположения и размеров операндов операции деления показаны в таблице 5.

Таблица 5. Расположение операндов и результата при делении

Делимое	**Делитель**	**Частное**	**Остаток**
16 бит в регистре ax	Байт регистр или ячейка памяти	Байт в регистре al	Байт в регистре ah
32 бит dx – старшая часть ax – младшая часть	Слово 16 бит регистр или ячейка памяти	Слово 16 бит в регистре ax	Слово 16 бит в регистре dx
64 бит edx – старшая часть eax – младшая часть	Двойное слово 32 бит регистр или ячейка памяти	Двойное слово 32 бит в регистре eax	Двойное слово 32 бит в регистре edx

После выполнения команды деления содержимое флагов неопределено, но возможно возникновение прерывания с номером 0, называемого “деление на ноль”. Этот вид прерывания относится к так называемым исключениям. Эта разновидность прерываний возникает внутри микропроцессора из-за некоторых аномалий во время вычислительного процесса.

Прерывание 0, “деление на ноль”, при выполнении команды div может возникнуть по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную под него разрядную сетку, что может случиться в следующих случаях:
- при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого в более чем 256 раз больше значения делителя;
- при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого в более чем 65 536 раз больше значения делителя;
- при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого в более чем 4 294 967 296 раз больше значения делителя.

Деление чисел со знаком. Для деления чисел со знаком предназначена команда

idiv делитель

Для этой команды справедливы все рассмотренные положения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения исключения 0, “деление на ноль”, в случае чисел со знаком. Оно возникает при выполнении команды idiv по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную для него разрядную сетку. Последнее в свою очередь может произойти:
- при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого в более чем 128 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –128 до +127);
- при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение делимого в более чем 32 768 раз больше значения делителя

(таким образом, частное не должно находиться вне диапазона от –32 768 до +32 768);

- при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого в более чем 2 147 483 648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от –2 147 483 648 до +2 147 483 647).

3.3 Логические команды

В системе команд микропроцессора есть следующий набор команд, поддерживающих работу с логическими данными:

and операнд_1,операнд_2 – операция логического умножения.

Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

or операнд_1,операнд_2 – операция логического сложения.

Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

xor операнд_1,операнд_2 – операция логического исключающего сложения.

Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

test операнд_1,операнд_2 – операция “проверить” (способом логического умножения). Команда выполняет поразрядно логическую операцию И над битами операндов операнд_1 и операнд_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

not операнд – операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

3.4 Команды сдвига

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем логические команды, рассмотренные выше. Все команды сдвига перемещают биты в поле операнда влево или вправо в зависимости от кода операции.

Количество сдвигаемых разрядов – счетчик_сдвигов – располагается, как видите, на месте второго операнда и может задаваться двумя способами:

- статически, что предполагает задание фиксированного значения с помощью непосредственного операнда;
- динамически, что означает занесение значения счетчика сдвигов в регистр cl перед выполнением команды сдвига.

Команды линейного сдвига делятся на два подтипа:

- команды логического линейного сдвига;
- команды арифметического линейного сдвига.

К командам логического линейного сдвига относятся:

- shl операнд,счетчик_сдвигов (Shift Logical Left) – логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули;
- shr операнд,счетчик_сдвигов (Shift Logical Right) – логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева (в позицию старшего, знакового бита) вписываются нули.

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

- sal операнд,счетчик_сдвигов (Shift Arithmetic Left) – арифметический сдвиг влево. Содержимое операнда

сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули. Команда sal не сохраняет знака, но устанавливает флаг cf в случае смены знака очередным выдвигаемым битом. В остальном команда sal полностью аналогична команде shl;

- sar операнд,счетчик_сдвигов (Shift Arithmetic Right) – арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева в операнд вписываются нули. Команда sar сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

- команды простого циклического сдвига;
- команды циклического сдвига через флаг переноса cf.

К командам простого циклического сдвига относятся:

- rol операнд,счетчик_сдвигов (Rotate Left) – циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые влево биты записываются в тот же операнд справа.
- ror операнд,счетчик_сдвигов (Rotate Right) – циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые вправо биты записываются в тот же операнд

К командам циклического сдвига через флаг переноса cf относятся следующие:

- rcl операнд,счетчик_сдвигов (Rotate through Carry Left) – циклический сдвиг влево через перенос. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса cf.

rcr операнд,счетчик_сдвигов (Rotate through Carry Right) – циклический сдвиг вправо через перенос. Содержимое операнда

сдвигается вправо на количество бит, определяемое операндом счетчик_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса cf.

1.5 Задание на выполнение

Выполните задания в соответствии с вашим вариантом. И подготовиться к ответам на теоретическую часть лабораторной работы.

Вариант 1
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A+B/2.
2. Установить все четные биты C.
3. Вывести на экран число C в двоичном виде.

Вариант 2
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A+B/3.
2. Обнулить все нечетные биты C.
3. Вывести на экран число C в двоичном виде.

Вариант 3
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A+B.
2. Если третий бит числа C установлен, то вывести на экран C в двоичном виде, в противном случае, вывести на экран C/2 в двоичном виде.

Вариант 4
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A/2+B.
2. Установить все нечетные биты C.

3. Вывести на экран число C в двоичном виде.

Вариант 5
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A+B)/4.
2. Сбросить первый бит числа C, если он установлен.
3. Вывести на экран число C в двоичном виде.

Вариант 6
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A-B)*4.
2. Выполнить циклический сдвиг полученного числа C на 3 бита вправо.
3. Вывести на экран число C в двоичном виде.

Вариант 7
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A/2+B.
2. Выполнить арифметический сдвиг C на 3 бит влево.
3. Вывести на экран число C в двоичном виде.

Вариант 8
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A+B*2.
2. Обнулить все четные биты C.
3. Вывести на экран число C в двоичном виде.

Вариант 9
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A+(B-5h)*2.

2. Если установлен четвертый бит числа C то вывести на экран A в двоичном виде, в противном случае вывести на экран число B в двоичном виде.

Вариант 10
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A+12h)/2+B.
2. Обнулить все четные биты C.
3. Вывести на экран число C в двоичном виде.

Вариант 11
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A-14h)*4-B.
2. Установить все четные биты C.
3. Вывести на экран число C в двоичном виде.

Вариант 12
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A*B-4.
2. Если третий и пятый бит числа C установлены, вывести на экран A в двоичном виде, если третий и пятый бит числа C сброшены, вывести на экран B в двоичном виде, в других случаях вывести на экран число C в двоичном виде.

Вариант 13
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A+B)/4-16.
2. Если третий и пятый бит числа C установлены, вывести на экран A в двоичном виде, если второй и четвертый бит числа C сброшены, вывести на экран B в двоичном виде, в других случаях вывести на экран число C в двоичном виде.

Вариант 14
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A-B)*2+1Ah.
2. Вывести на экран 1 если шестой бит числа C установлен и 0 в противном случае.

Вариант 15
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A-7h)*4-B.
2. Сбросить третий бит числа C, если он установлен.
3. Вывести на экран число C в двоичном виде.

Вариант 16
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A+4*B.
2. Выполнить циклический сдвиг числа C на 3 бита вправо.
3. Вывести на экран число C в двоичном виде.

Вариант 17
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A*3+B*2.
2. Выполнить арифметический сдвиг числа C на 2 бита вправо.
3. Вывести на экран число C в двоичном виде.

Вариант 18
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A*3+B*2.
2. Выполнить арифметический сдвиг числа C на 2 бита вправо.

3. Вывести на экран число C в двоичном виде.

Вариант 19
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=A*B-4.
2. Если третий и пятый бит числа C установлены, вывести на экран A в двоичном виде, если третий и пятый бит числа C сброшены, вывести на экран B в двоичном виде, в других случаях вывести на экран число C в двоичном виде.

Вариант 20
Пользователь вводит два числа A и B в шестнадцатеричном виде. Программа должна:
1. Посчитать C=(A+17h)/2+B.
2. Обнулить все четные биты C.
3. Вывести на экран число C в двоичном виде.

ЛАБОРАТОРНАЯ РАБОТА №4
«Модульное программирование»

4.1 Цель работы

Целью данной работы является изучение написания процедур и макросов с использованием модульной структуры программы.

4.2 Процедуры на языке ассемблера

Для оформления процедур существуют специальные директивы proc/endp и машинная команда ret – возврат управления из процедуры вызывающей программе.

Формат команды ret: ret число.

Работа команды зависит от типа процедуры:

- для процедур ближнего типа – восстановить из стека содержимое eip/ip;
- для процедур дальнего типа – последовательно восстановить из стека содержимое eip/ip и сегментного регистра cs.
- если команда ret имеет операнд, то увеличить содержимое esp/sp на величину операнда число; при этом учитывается атрибут режима адресации – use16 или use32:
  - если use16, то sp=(sp+число), то есть указатель стека сдвигается на число байт, равное значению число;
  - если use32, то sp=(sp+2*число), то есть указатель стека сдвигается на число слов, равное значению число.

Процедуры могут размещаться в программе:

- в начале программы (до первой исполняемой команды);
- в конце программы (после команды выхода в операционную систему);
- промежуточный вариант – тело процедуры располагается внутри другой процедуры или основной программы (с использованием команды безусловного перехода jmp);
- в другом модуле.

Вызов близкой или дальней процедуры с запоминанием в стеке адреса точки возврата осуществляется *командой call* (формат: call метка). Выполнение команды не влияет на флаги.

При ближней адресации – в стек заносится содержимое указателя команд eip/ip и в этот же регистр загружается новое значение адреса, соответствующее метке;

При дальней адресации – в стек заносится содержимое указателя команд eip/ip и cs. Затем в эти же регистры загружаются новые значения адресов, соответствующие дальней метке;

4.3 Передача аргументов через регистры

Существуют следующие варианты передачи аргументов в процедуру (модуль):

- *Через регистры*. Данные становятся доступными немедленно после передачи управления процедуре. Очень популярный способ при небольшом объеме передаваемых данных.
- *Через общую область памяти*. Необходимо использовать атрибут комбинирования сегментов – common (см. лаб. работу 1). Сегменты будут перекрываться в памяти и, следовательно, совместно использовать выделенную память.
- *Через стек*. Наиболее часто используемый способ. Суть его в том, что вызывающая процедура самостоятельно заносит в стек передаваемые данные, после чего производит вызов вызываемой процедуры.
- *С помощью директив extrn и public*. Директива extrn предназначена для объявления некоторого имени внешним по отношению к данному модулю. Это имя в другом модуле должно быть объявлено в директиве public. Директива public предназначена для объявления некоторого имени, определенного в этом модуле и видимого в других модулях.

  Синтаксис директив:

```
Extrn имя:тип, …, имя:тип
Public имя, …, имя
```

Здесь имя – идентификатор, определенный в другом модуле. В качестве идентификатора могут выступать:

- имена переменных (определенных операторами db, dw и т.д.);
- имена процедур;
- имена констант (определенных операторами = и equ).

Возможные значения типа определяются допустимыми типами объектов для этих директив:

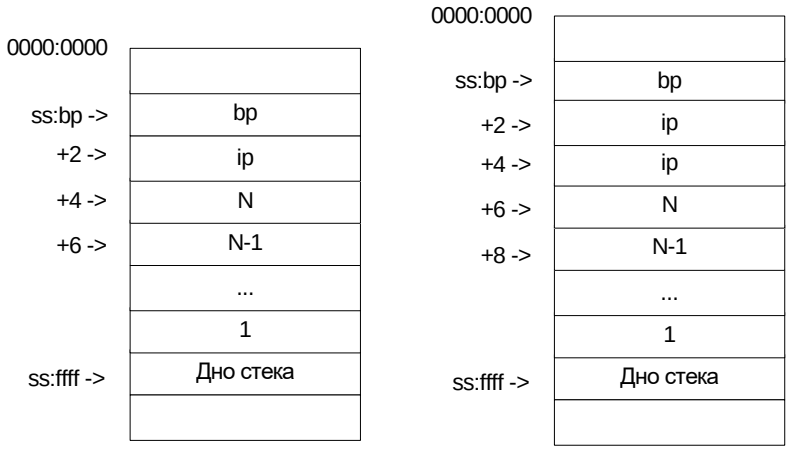
- если имя – это переменная, то тип может принимать значения byte, word, dword, pword, fword, qword и tbyte;
- если имя – это процедура, то тип может принимать значения near или far;
- если имя – это константа, то тип должен быть abs.

Остановимся боле подробно при передаче параметров через стек. Приведем пример структуры вызываемой процедуры при ближней адресации.

```
asmpr proc  near
;пролог
 push bp
 mov  bp,sp
. . .
;доступ к элементам стека
 mov ax, [bp+4] ;доступ к N-элементу
 mov ax, [bp+6] ;доступ к N-1-элементу
 mov ax, [bp+8] ;доступ к N-2-элементу
. . .
; эпилог
 mov  sp,bp ;восстановление sp
 pop  bp ;восстановление bp

 ret   ;возврат в вызывающую программу
asmpr endp  ;конец процедуры
```

На рис. 1. показана структура стека при ближней (а) и дальней (б) адресации в нутрии вызываемой процедуры.



а) б)

Рис. 1. Структура стека в вызываемой процедуре при использовании различных видов адресации: а – ближняя адресация; б – дальняя адресация.

При использовании дальней адресации для доступа к элементам стека требуется поправка на +2.

```
;доступ к элементам стека при дальней адресации
    mov ax, [bp+6] ;доступ к N-элементу
    mov ax, [bp+8] ;доступ к N-1-элементу
    mov ax, [bp+10] ;доступ к N-2-элементу
```

4.4 Возврат результата из процедуры

Существует три варианта возврата результата из процедуры:

- С использованием регистров.
- С использованием общей памяти.
- С использованием стека. Здесь возможны два варианта:
  - Использование для возвращаемых аргументов тех же ячеек в стеке, которые применялись для передачи аргументов в процедуру.

- Предварительное помещение в стек наряду с передаваемыми аргументами фиктивных аргументов с целью резервирования места для возвращаемого значения.

4.5 Макросредства языка ассемблера

4.5.1 Псевдооператоры equ и =

К простейшим макросредствам языка ассемблера можно отнести псевдооператоры equ и "=" (равно). Эти псевдооператоры предназначены для присвоения некоторому выражению символического имени или идентификатора. Впоследствии, когда в ходе трансляции этот идентификатор встретится в теле программы, макроассемблер подставит вместо него соответствующее выражение. В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Синтаксис псевдооператора equ:

имя_идентификатора equ строка или числовое_выражение

Синтаксис псевдооператора “=”:

имя_идентификатора = числовое_выражение

Несмотря на внешнее и функциональное сходство псевдооператоры equ и “=” отличаются следующим:

- из синтаксического описания видно, что с помощью equ идентификатору можно ставить в соответствие, как числовые выражения, так и текстовые строки, а псевдооператор “=” может использоваться только с числовыми выражениями;
- идентификаторы, определенные с помощью “=”, можно переопределять в исходном тексте программы, а определенные с использованием equ – нельзя.

Ассемблер всегда пытается вычислить значение строки, воспринимая ее как выражение. Для того, чтобы строка

воспринималась именно как текстовая, необходимо заключить ее в угловые скобки: <строка>. Кстати сказать, угловые скобки являются оператором ассемблера, с помощью которого транслятору сообщается, что заключенная в них строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы. Хотя в режиме Ideal это не обязательно, так как строка для equ в нем всегда трактуется как текстовая.

Псевдооператор **equ** удобно использовать для настройки программы на конкретные условия выполнения, замены сложных в обозначении объектов, многократно используемых в программе более простыми именами и т. п.

Псевдооператор "=" удобно использовать для определения простых абсолютных (то есть независящих от места загрузки программы в память) математических выражений. Главное условие то, чтобы транслятор мог вычислить эти выражения во время трансляции.

4.5.2 Макрокоманды

Идейно макрокоманда представляет собой дальнейшее развитие механизма замены текста. С помощью макрокоманд в текст программы можно вставлять последовательности строк (которые логически могут быть данными или командами) и даже более того – привязывать их к контексту места вставки.

Макрокоманда представляет собой строку, содержащую некоторое символическое имя – имя макрокоманды, предназначенное для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами.

Обычно программист сам чувствует момент, когда ему нужно использовать макрокоманды в своей программе. Если такая необходимость возникает, и нет готового ранее разработанного варианта нужной макрокоманды, то вначале необходимо задать ее шаблон-описание, который называют макроопределением.

Синтаксис макроопределения следующий:

```
имя_макрокоманды macro список_аргументов
 тело макроопределения
endm
```

Есть три варианта размещения макроопределения:

- В начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. Этот вариант следует применять в случаях, если определяемые вами макрокоманды актуальны только в пределах одной этой программы.
- В отдельном файле. Этот вариант подходит при работе над несколькими программами одной проблемной области. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву include имя_файла.
- В макробиблиотеке. Если у вас есть универсальные макрокоманды, которые используются практически во всех ваших программах, то их целесообразно записать в так называемую макробиблиотеку. Сделать актуальными макрокоманды из этой библиотеки можно с помощью все той же директивы include.

Если в программе некоторая макрокоманда вызывается несколько раз, то в процессе макрогенерации возникнет ситуация, когда в программе один идентификатор будет определен несколько раз, что, естественно, будет распознано транслятором как ошибка. Для выхода из подобной ситуации применяют директиву **local**, которая имеет следующий синтаксис: local список_идентификаторов.

4.6. Задание на выполнение

Модифицировать предыдущую программу в виде:

Вариант 1

1. Написать процедуру для вывода результата
2. Написать процедуру для ввода чисел
3. Написать макрос для расчета

Передача параметров через регистры

Вариант 2
1. Написать процедуру для вывода результата
2. Написать макрос для ввода чисел
3. Написать макрос для расчета
Передача параметров через стек

Вариант 3
1. Написать макрос для вывода результата
2. Написать процедуру для ввода чисел
3. Написать макрос для расчета
Передача параметров через стек

Вариант 4
1. Написать макрос для вывода результата
2. Написать макрос для ввода чисел
3. Написать процедуру для расчета
Передача параметров через стек

Вариант 5
1. Написать процедуру для вывода результата
2. Написать процедуру для ввода чисел
3. Написать макрос для расчета
Передача параметров через стек

Вариант 6
1. Написать процедуру для вывода результата
2. Написать макрос для ввода чисел
3. Написать процедуру для расчета
Передача параметров через стек

Вариант 7
1. Написать макрос для вывода результата
2. Написать процедуру для ввода чисел
3. Написать процедуру для расчета

Передача параметров через стек

Вариант 8
1. Написать процедуру для вывода результата
2. Написать процедуру для ввода чисел
3. Написать процедуру для расчета
Передача параметров через стек

Вариант 9
1. Написать макрос для вывода результата
2. Написать макрос для ввода чисел
3. Написать макрос для расчета
Передача параметров через общую память

Вариант 10
1. Написать процедуру для вывода результата
2. Написать макрос для ввода чисел
3. Написать макрос для расчета
Передача параметров через общую память

Вариант 11

1. Написать макрос для вывода результата
2. Написать процедуру для ввода чисел
3. Написать макрос для расчета
Передача параметров через общую память

Вариант 12
1. Написать макрос для вывода результата
2. Написать макрос для ввода чисел
3. Написать процедуру для расчета
Передача параметров через общую память

Вариант 13
1. Написать процедуру для вывода результата
2. Написать процедуру для ввода чисел
3. Написать макрос для расчета

Передача параметров через общую память

Вариант 14
1. Написать макрос для вывода результата
2. Написать процедуру для ввода чисел
3. Написать процедуру для расчета
Передача параметров через общую память

Вариант 15

1. Написать процедуру для вывода результата
2. Написать макрос для ввода чисел
3. Написать процедуру для расчета
Передача параметров через общую память

Вариант 16
1. Написать процедуру для вывода результата
2. Написать процедуру для ввода чисел
3. Написать процедуру для расчета
Передача параметров через общую память

Вариант 17
1. Написать макрос для вывода результата
2. Написать макрос для ввода чисел
3. Написать макрос для расчета
Передача параметров через регистры

Вариант 18
1. Написать процедуру для вывода результата
2. Написать макрос для ввода чисел
3. Написать макрос для расчета
Передача параметров через регистры

Вариант 19
1. Написать макрос для вывода результата
2. Написать процедуру для ввода чисел
3. Написать макрос для расчета

Передача параметров через регистры

Вариант 20
1. Написать макрос для вывода результата
2. Написать макрос для ввода чисел
3. Написать процедуру для расчета
Передача параметров через регистры

ЛАБОРАТОРНАЯ РАБОТА №5 «Совершенствование навыков работы на языке ассемблера»

5.1 Цель работы

Целью работы является совершенствование навыков работы на языке ассемблера.

5.2 Задание на выполнение

Введите матрицу из N,N (или массив из N) элементов на языке и выполните одно из действий согласно варианту.

Вариант 1.
Выбрать все элементы главной диагонали матрицы.

Вариант 2.
Найти максимальный и минимальный элемент матрицы.

Вариант 3.
Построить обратную матрицу.

Вариант 4.
Из всех элементов матрицы вычесть минимальный элемент.

Вариант 5.
Обнулить в матрице все четные числа

Вариант 6.
Обнулить в матрице все нечетные числа.

Вариант 7.
Поменять в матрице четные столбцы с нечетными.

Вариант 8.
Поменять в матрице четные строки с нечетными.

Вариант 9.
Умножить элементы матрицы на минимальный элемент.

Вариант 10.
Найти среднеарифметическое число всех элементов матрицы (целая часть, остаток)

Вариант 11.
Найти определитель матрицы 3*3.

Вариант 12.
Сложить все элементы матрицы построчно с элементом на главной диагонали текущей строки.

Вариант 13.
Сложить все элементы матрицы по столбцам с элементом на главной диагонали текущего столбца.

Вариант 14.
Посчитать сумму элементов всех строк.

Вариант 15.
Посчитать сумму элементов всех столбцов.

Вариант 16.
Посчитать сумму элементов главной диагонали

Вариант 17.
Посчитать сумму элементов обратной диагонали.

Вариант 18.
Ввести два массива и посчитать сумму и разность соответствующих элементов массивов.

Вариант 19.

Ввести два массива и выполнить действия по умножению и делению каждого элемента с одинаковым индексом (при делении запоминать только целую часть).

Вариант 20.

Ввести два массива и найти максимальные элементы. Сравнить их. Вывести тот массив, у которого максимальный элемент меньше, чем максимальный элемент другого массива.

ЛАБОРАТОРНАЯ РАБОТА №6 «Интерфейс с языками высокого уровня и обработка массивов»

6.1 Цель работы

Целью работы является изучение связи подпрограмм на ассемблере с программами, написанными на языках высокого уровня и обработка массивов на языке ассемблере.

6.2 Формы комбинирования программ на языках высокого уровня с ассемблером

Существуют следующие формы комбинирования программ:

- Использование ассемблерных вставок. Эта форма сильно зависит от синтаксиса языка высокого уровня и конкретного компилятора.
- Использование внешних процедур и функций (на уровне объектных модулей). Это более универсальная форма комбинирования. Она имеет ряд преимуществ:
  - Написание и отладку программ можно производить независимо.
  - Написанные программы можно использовать в других проектах.
  - Облегчается модификация и сопровождение программ в течение жизненного цикла проекта.

Разработка таких подпрограмм на языке ассемблера требует ясного представления о том, каким образом взаимодействуют подпрограммы в разных языках (таблица 6). Передача аргументов, как правило, осуществляется через стек.

Таблица 6. Сравнение механизмов взаимодействия подпрограмм

	Си	Паскаль
Тип подпрограммы	Функция	Процедура или функция

Направление передачи аргументов	Справа налево	Слева направо
Аргументы передаются	По значению (адрес это указатель)	По значению и по адресу
Процедура чистящая стек	Вызывающая	Вызываемая

6.3 Соглашения о связях для языка Си

Если в программе на языке Си используется обращение к внешнему модулю, написанному на другом языке, необходимо включить в программу прототип внешней функции (описание точки входа), например:

```
extern "C" void asmpoc(char ch, unsigned x, unsigned y)
```

Функция asmpoc должна описываться на ассемблере следующим процедурным блоком:

```
public _asmproc
 _asmproc proc ...
 ...
 _asmproc endp
```

Блок extrn "C" добавляет к имени точки входа функции префикс (символ подчеркивания). Некоторые компиляторы Си не требуют наличия "C", символ подчеркивания добавляется по умолчанию.

Тип ассемблерной подпрограммы зависит от используемой компилятором модели памяти. Модели памяти и типы указателей на подпрограммы сведены в таблице 7.

Таблица 7. Соотношение моделей памяти и типов указателей

Модель	Ключ BCC-компилятора	Указатель на функцию	Указатель на данные
Small	-ms	near, CS	near, DS
Compact	-mc	far	near, DS
Medium	-mm	near, CS	far
Large	-ml	far	far

Для малой модели памяти (Small) сегмент кода ассемблерной подпрограммы объединяется с кодовым сегментом главной программы на СИ, который для малой модели всегда имеет имя _TEXT. Это же имя должен иметь сегмент кода ассемблерной внешней подпрограммы или используйте упрощенные директивы сегментации, где нет необходимости указывать имя сегмента

Для большой модели памяти (Large) сегмент кода в ассемблерной подпрограмме не объединяется и может иметь любое имя.

5.4 Соглашение о связях для языка Паскаль

В программу на языке Паскаль необходимо включить прототип (описание) точки входа во внешнюю подпрограмму или функцию по правилам описания заголовка процедуры или функции:

```
Procedure
asmproc(ch:char;x,y,kol:integer); external;
```

Директива EXTERNAL указывает, что описание подпрограммы не содержится в главной программе. Для указания файла, содержащего объектный модуль внешней подпрограммы, в программе на Паскале используется директива {$L путь_до_объектного_модуля}.

Компоновщик объединяет этот сегмент с сегментом главной программы.

Занесение аргументов в стек при вызове подпрограммы производится в порядке следования аргументов в списке прототипа. При передаче по значению сами аргументы заносятся в стек, даже массивы.

По умолчанию компилятор Паскаля вставляет в программу команду CALL дальнего вызова. Если необходимо организовать ближний вызов, то перед прототипом внешней подпрограммы следует поставить директиву {$F-} - отмена дальнего вызова. Эта директива действует только на одну подпрограмму и формирует для нее CALL ближнего обращения. Директива не действует на адреса аргументов - в стек заносится полный адрес.

6.4 Задание на выполнение

1. Введите матрицу из N,N (или массив из N, согласно предыдущей лабораторной работы) элементов на языке Си или Паскаль.
2. Передайте его в качестве аргументов в процедуру языка Ассемблера.
3. Выполните одно из действий (согласно предыдущей лабораторной работы).
4. Результат передайте в вызывающую программу и выведите на печать.

ЛАБОРАТОРНАЯ РАБОТА №7
«Использование цепочечных команд»

7.1 Цель работы

Целью работы является изучение работы цепочечных команд при обработке массивов на языке ассемблера.

7.2 Цепочечные команды

Пересылка цепочек в память

```
movs приемник,источник
movsb
movsw
movsd
```

Алгоритм работы:

Выполнить копирование байта, слова или двойного слова из операнда источника в операнд приемник, при этом адреса элементов предварительно должны быть загружены:

- адрес источника — в пару регистров ds:esi/si (ds по умолчанию, допускается замена сегмента);

- адрес приемника — в пару регистров es:edi/di (замена сегмента не допускается);

В зависимости от состояния флага df изменить значение регистров esi/si и edi/di:

- если df=0, то увеличить содержимое этих регистров на длину структурного элемента последовательности;

- если df=1, то уменьшить содержимое этих регистров на длину структурного элемента последовательности;

Если есть префикс повторения, то выполнить определяемые им действия.

Пример программы пересылки цепочки в память:

```
.data
source  db      'Тестируемая строка','$'      ;строка-источник
dest    db      19 DUP (' ')   ;строка-приёмник
.code
```

```
        assume ds:@data,es:@data
main:   mov    ax,@data       ;загрузка сегментных регистров
        mov    ds,ax          ;настройка регистров DS и ES
        mov    es,ax          ;на адрес сегмента данных
        cld    ;сброс флага DF – обработка строки от начала к
концу
        lea    si,source      ;загрузка в SI смещения строки-
источника
        lea    di,dest ;загрузка в DI смещения строки-приёмника
        mov    cx,20   ;для префикса rep – счетчик повторений
rep     movs   dest,source    ;пересылка строки

        lea    dx,dest
        mov    ah,09h  ;вывод на экран строки-приёмника
        int    21h
exit:
```

Сравнение цепочек в памяти.

```
cmps приемник,источник
cmpsb
cmpsw
cmpsd
```

Алгоритм работы:

- выполнить вычитание элементов (источник - приемник), адреса элементов предварительно должны быть загружены:
  - адрес источника — в пару регистров ds:esi/si;
  - адрес назначения — в пару регистров es:edi/di;
- в зависимости от состояния флага df изменить значение регистров esi/si и edi/di:
  - если df=0, то увеличить содержимое этих регистров на длину элемента последовательности;
  - если df=1, то уменьшить содержимое этих регистров на длину элемента последовательности;
- в зависимости от результата вычитания установить флаги:
  - если очередные элементы цепочек не равны, то cf=1, zf=0;
  - если очередные элементы цепочек или цепочки в целом равны, то cf=0, zf=1;
- при наличии префикса выполнить определяемые им действия.

Пример программы сравнения цепочек в памяти:

```
.data
obl1    db      'Строка для сравнения'
obl2    db      'Строка для сравнения'
a_obl1  dd      obl1
a_obl2  dd      obl2
.code
  ...
  cld   ;просмотр   цепочки   в   направлении   возрастания
адресов
  mov     cx,20   ;длина цепочки
  lds      si,a_obl1        ;адрес источника в пару ds:si

  les     di,a_obl2        ;адрес назначения в пару es:di
  repe    cmpsb            ;сравнивать, пока равны
  jnz      m1       ;если не конец цепочки, то встретились
разные элементы ...                        ;действия,  если
цепочки совпали ...
  m1:
  ...                        ;действия,   если   цепочки   не
совпали
```

Сканирование цепочек в памяти

scas приемник

scasb

scasw

scasd

Алгоритм работы:

- выполнить вычитание (элемент цепочки-(eax/ax/al)). Элемент цепочки локализуется парой es:edi/di. Замена сегмента es не допускается;
- по результату вычитания установить флаги;
- изменить значение регистра edi/di на величину, равную длине элемента цепочки. Знак этой величины зависит от состояния флага df:

  df=0 — величина положительная, то есть просмотр от начала цепочки к ее концу;

  df=1 — величина отрицательная, то есть просмотр от конца цепочки к ее началу.

Пример программы сканирования цепочек в памяти:

```
;сосчитать число пробелов в строке str
.data
str     db      '...'
```

```
len_str=$-str
.code
 mov     ax,@data
 mov     ds,ax
 mov     es,ax
 lea       di,str
 mov     cx,len_str      ;длину строки — в cx
 mov     al,' '
 mov     bx,0    ;счетчик для подсчета пробелов в строке

 cld
cycl:
repe     scasb
 jcxz      exit      ;переход   на   exit,   если   цепочка
просмотрена полностью
 inc     bx
 jmp     cycl
exit:   ...
```

Загрузка элемента цепочки в аккумулятор

```
lods источник
lodsb
lodsw
lodsd
```

Алгоритм работы:

- загрузить элемент из ячейки памяти, адресуемой парой ds:esi/si, в регистр al/ax/eax. Размер элемента определяется неявно (для команды lods) или явно в соответствии с применяемой командой (для команд lodsb, lodsw, lodsd);
- изменить значение регистра si на величину, равную длине элемента цепочки. Знак этой величины зависит от состояния флага df:
  - df=0 — значение положительное, то есть просмотр от начала цепочки к ее концу;
  - df=1 — значение отрицательное, то есть просмотр от конца цепочки к ее началу.

Загрузка элемента из аккумулятора в цепочку

```
stos приемник
stosb
stosw
stosd
```

Алгоритм работы:

- записать элемент из регистра al/ax/eax в ячейку памяти, адресуемую парой es:di/edi. Размер элемента определяется неявно (для команды stos) или конкретной применяемой командой (для команд stosb, stosw, stosd);
- изменить значение регистра di на величину, равную длине элемента цепочки. Знак этого изменения зависит от состояния флага df:
  - df=0 — увеличить, что означает просмотр от начала цепочки к ее концу;
  - df=1 — уменьшить, что означает просмотр от конца цепочки к ее началу.

Пример совместной работы stosb и lodsb:

```
;копировать одну строку в другую до первого пробела
str1    db      'Какая-то строка'
len_str1=$-str
str2    db    len_str1 dup (' ')
...
 mov     ax,@data
 mov     ds,ax
 mov     es,ax
 cld
 mov     cx,len_str1
 lea     si,str1
 lea     di,str2
m1:             lodsb
 cmp     al,' '
 jc      exit    ;выход, если пробел
 stosb
 loop    m1
exit:
```

7.3 Задание на выполнение

Реализовать задание из предыдущей лабораторной работы №5 полностью на языке ассемблера с применением цепочечных команд.

ЛАБОРАТОРНАЯ РАБОТА №8 «Программирование FPU»

Выполняется в течении двух лабораторных работ.

8.1 Цель работы

Целью работы является изучение работы команд устройства с плавающей арифметикой на языке ассемблера.

8.2 Организация FPU

Общее положение

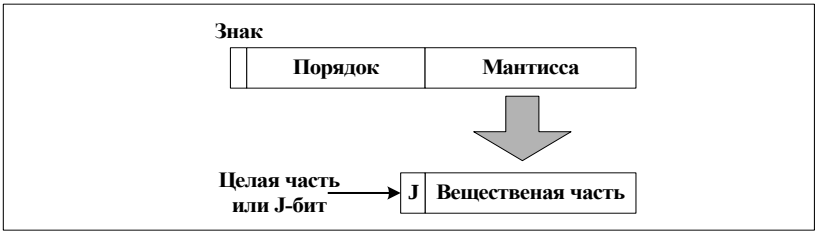
Устройство плавающей арифметики Intel-архитектуры предоставляет возможность высокопроизводительных вычислений. Оно поддерживает вещественные, целые и BCD целые типы данных, алгоритмы вещественной арифметики и архитектуру обработки исключение определенные в стандартах IEEE 754 и 854.

Intel-архитектура FPU развивалась параллельно с Intel-архитектурой ранних процессоров. Первые математические сопроцессоры (Intel 8087, Intel 287 и Intel 387) были дополнительными устройствами к Intel 8086/8088, Intel 286 и Intel 386 процессорам соответственно.

Начиная с Intel 486 DX процессора, устройство плавающей арифметики помещается на один чип с самим процессором.

Формат чисел с плавающей точкой

Для увеличения скорости и эффективности вычислений, компьютеры или FPU обычно представляют вещественные числа в двоичном формате с плавающей точкой. В этом формате вещественное число имеет три части: знак, мантиссу и порядок.



Знак – это двоичное значение, которое определяет либо число положительное (0), либо число отрицательное (1).

Мантисса имеет две части: 1 битовое целое число (известное как J-бит) и двоичная вещественная часть. J-бит обычно не присутствует, а имеет фиксированное значение.

Порядок – это двоичное целое число, определяющее степень, в которую необходимо возвести 2.

Среда выполнения инструкций FPU состоит из 8 регистров данных и следующих регистров специального назначения:

- Регистр статуса.
- Регистр состояния.
- Регистр слова тега.
- Регистра указателя инструкции.
- Регистра последнего операнда.
- Регистра кода команды.

Регистры данных FPU состоят из восьми 80 битовых регистров. Значения хранятся в этих регистрах в расширенном вещественном формате. Когда вещественные, целые или BCD целые значения загружаются из памяти в любой из регистров данных FPU, значения автоматически конвертируются в

расширенный вещественный формат. Когда результат вычислений перемещается назад в память, он конвертируется в один из типов данных FPU

FPU инструкции обращаются к регистрам данных как к регистрам стека. Все адресации к регистрам данных происходят относительно регистра на вершине стека. Номер регистра на вершине стека находится в поле TOP слова статуса FPU. Операция загрузки уменьшает TOP на 1 и загружает значение в новый регистр на вершине стека. Операция сохранения сохраняет содержимое регистра на вершине стека и увеличивает TOP на 1.

Если операция загрузки выполняется, когда TOP равно 0, происходит циклический возврат и новое значение TOP становится равно 7. Исключение переполнения стека плавающей арифметики происходит для индикации того, что произошел циклический возврат и не сохраненное значение может быть перезаписано.

Многие инструкции плавающей арифметики имеют несколько режимов адресации, что позволяет программисту неявно оперировать на вершине стека или явно оперировать с регистрами относительно вершины стека.

Пример исследования команд передачи данных в FPU:

```
.586p
masm
model    use16 small
.stack   100h
.data    ;сегмент данных
ch_dt    dt      43567  ;ch_dt=00 00 00 00 00 00 00 04 35 67
x dw     3       ;x=00 03
y_real   dq      34e7   ;y_real=41 b4 43 fd 00 00 00 00
ch_dt_st dt      0
x_st     dw      0
y_real_st        dq      0
.code
main     proc    ;начало процедуры main
 mov     ax, @data
 mov     ds, ax
 fbld    ch_dt   ;st(0)=43567
 fild    x       ;st(1)=43567, st(0)=3
 fld     y_real  ;st(2)=43567, st(1)=3, st(0)=340000000
 fxch    st(2)   ;st(2)=340000000, st(1)=3, st(0)=43567
```

```
  fbstp   ch_dt_st       ;st(1)=340000000,  st(0)=3  ch_dt_st=00  00
  00 00 00 00 00 04 35 67
  fistp   x_st    ;st(0)=340000000, x_st=00 03
  fstp    y_real_st      ;y_real_st=41 b4 43 fd 00 00 00 00
exit:     mov     ax, 4c00h
  int     21h
main      endp
end       main
```

Пример вычисление выражения z=(sqrt(|x|)-y)^2:

```
  .586p
  masm
  model   use16 small
  .stack  100h
  .data   ;сегмент данных
  ;исходный данные:
  x       dd      -29e-4
  y       dq      4.6
  z       dd      0
  .code
  main    proc
          mov     ax,@data
          mov     ds,ax
          finit   ;приведение сопроцессора в начальное состояние
          fld     x       ;st(0)=x
          fabs            ;st(0)=|x|
          fsqrt
          fsub    y       ;st(0)=sqrt|x|-y
          fst     st(1)
          fmul
          fst     z
  exit:   mov     ax,4c00h
          int     21h
  main    endp
  end     main
```

8.3 Задание на выполнение

Вычислите выражение согласно варианту:

Вариант 1.
Z=5.3*X^2+7.2*Y+2.8

Вариант 2.
Z=5.3+ sqrt(|X|)/Y

Вариант 3.
Z=sqrt(|X|)+Y^3

Вариант 4.
Z=X*Y/3.4

Вариант 5.
Z=X+Y/(|X-Y|)

Вариант 6.
Z=2.1*X^Y

Вариант 7.
Z=4.4*(X+Y)

Вариант 8.
Z=sqtr(|X-Y|)*3.3

Вариант 9.
Z=X^3-Y^2

Вариант 10.
Z=|X*Y/4.3|

Вариант 11.
Z=sqtr(X)-sqrt(Y)

Вариант 12.
Z=X*Y-X/Y

Вариант 13.
Z=(X-1.4)*(Y-3.1)

Вариант 14.
Z=(X+2.5)/(Y+0.3)

Вариант 15.

Z=|X|/|Y|+X*Y

Вариант 16.
Z=|X/Y|+|X*Y|

Вариант 17.
Z=sqrt(|X*Y|)

Вариант 18.
Z=sqrt(|X/Y|)

Вариант 19.
Z=sqrt(|X|)+Y^2

Вариант 20.
Z=X^Y+Y^X

Методические указания к самостоятельной работе

1. Подготовка к опросам на лекции 1**8 часов** (Обзор архитектур современных процессоров – 6 часов, программирование на языке Ассемблера в реальном режиме – 4 часа, связь языка Ассемблера с языками высокого уровня – 4 часа, программирование на языке Ассемблера в защищенном режиме – 4 часа).

2. Подготовка к лабораторным работам и оформление отчетов по ЛР из расчета 1 час на 1 час ЛР (**36 часов** самостоятельной работы).

Для подготовки к лабораторным работам следует использовать данные методические указания, в качестве дополнительной литературы следует воспользоваться литературой [1-3], приведенной в разделе «Список литературы»

Форма контроля: Допуск к лаб. работам. Защита отчета по ЛР.

Подготовка к экзамену **36 часов.**

Итого самостоятельной работы – **90 часа**.

Список литературы

1. Assembler / В. Юров. – СПб, 2001. – 624 с.: ил.
2. www.wasm.ru
3. www.xakep.ru