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")
Заочная форма обучения, план набора 2012 г.

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

«____» _____ 2017г.

Томск – 2017

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.

Задача, решаемая дисциплиной формирование компетенции: владение архитектурой электронных вычислительных машин и систем.

Дисциплина «Архитектура вычислительных систем» (Б1.Б.15) относится к блоку 1 (базовая часть).

Предшествующими дисциплинами, формирующими начальные знания, являются следующие дисциплины: Вычислительные системы, сети и телекоммуникации, Операционные системы и сети.

Процесс изучения дисциплины направлен на формирование 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, изучение связи подпрограмм на ассемблере с программами, написанными на языках высокого уровня и обработка массивов на языке ассемблере.

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 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

1.5 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.

1.6 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.

Трудоемкость лабораторной работы: 4 часа.

ЛАБОРАТОРНАЯ РАБОТА № 2 «Программирование FPU»

Выполняется в течении двух лабораторных работ.

2.1 Цель работы

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

2.2 Организация FPU

Общее положение

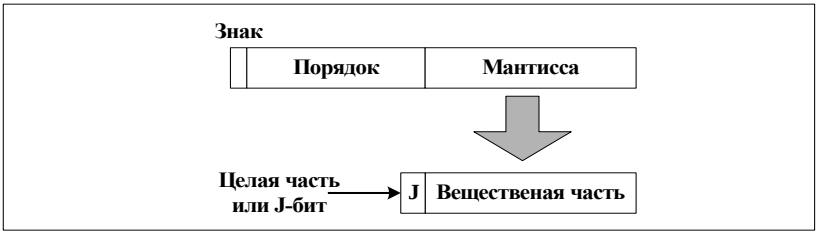
Устройство плавающей арифметики Intel-архитектуры предоставляет возможность высокопроизводительных вычислений. Оно поддерживает вещественные, целые и BCD целые типы данных, алгоритмы вещественной арифметики и архитектуру обработки исключение определенные в стандартах IEEE 754 и 854.

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
```

2.3 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

Трудоемкость лабораторной работы: 4 часа.

Методические указания к самостоятельной работе

4 семестр.

Проработка лекционного материала:

Обзор архитектур современных процессоров – 18 часов.

Программирование на языке Ассемблера в реальном режиме – 50 часов.

6 семестр.

Проработка лекционного материала:

Связь языка Ассемблера с языками высокого уровня – 20 часов.

Программирование на языке Ассемблера в защищенном режиме – 20 часов.

Подготовка к лабораторным работам и оформление отчетов по лабораторным работам из расчета 1 час на 1 час лабораторной работы (8 часов самостоятельной работы).

Выполнение контрольных работ – 3 часа.

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

Подготовка и сдача экзамена – 9 часов.

Итого самостоятельной работы – **128 часов.**

Список литературы

1. Вычислительные системы, сети и телекоммуникации: Учебное пособие / Гриценко Ю. Б. - 2015. 134 с.: Научно-образовательный портал ТУСУР, https://edu.tusur.ru/publications/5053
2. Системное программное обеспечение: Учебное пособие / Гриценко Ю. Б. - 2006. 174 с.: Научно-образовательный портал ТУСУР, http://edu.tusur.ru/publications/635
3. Assembler / В. Юров. – СПб, 2001. – 624 с.: ил.