

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Архитектура вычислительных систем

Лабораторные работы

Учебное пособие

Москва

Российский университета дружбы народов

2019

УДК 004.2 (075.8)
ББК 32.973.26-02
А 87

Утверждено
РИС Учёного совета
Российского университета
дружбы народов

Рецензенты:

доктор технических наук, профессор, начальник отдела УИТО и СТС
РУДН *К. Е. Самуилов*;
доцент, кандидат физико-математических наук, с.н.с. ЛИТ ОИЯИ
О. И. Стрельцова

Авторский коллектив:

**А. В. Демидова, Т. Р. Велиева, М. Н. Геворкян,
А. В. Королькова, Д. С. Кулябов.**

А-87 Архитектура вычислительных систем: лабораторные работы :
учебное пособие / А. В. Демидова, Т. Р. Велиева, М. Н. Геворкян,
А. В. Королькова, Д. С. Кулябов. — Москва : РУДН, 2019. — 87 с. :
ил.

Данное учебное пособие рекомендуется для проведения лабораторных работ по курсу «Архитектура вычислительных систем» для направлений 02.03.02 «Фундаментальная информатика и информационные технологии», 09.03.03 «Прикладная информатика», 38.03.05 «Бизнес-информатика» и по курсу «Архитектура компьютеров» для направлений 01.03.02 «Прикладная математика и информатика», 02.03.01 «Математика и компьютерные науки».

УДК 004.2(075.8)
ББК 32.973.26-02

ISBN 978-5-209-08880-6

© Демидова А. В., Велиева Т. Р.,
Геворкян М. Н., Королькова А. В.,
Кулябов Д. С., 2019
© Российский университет дружбы народов,
2019

Содержание

| | |
|--|-----------|
| Лабораторная работа № 1. Основы интерфейса командной строки ОС GNU Linux | 5 |
| 1.1. Цель работы | 5 |
| 1.2. Предварительные сведения | 5 |
| 1.3. Порядок выполнения работы | 15 |
| 1.4. Содержание отчёта | 15 |
| 1.5. Контрольные вопросы | 16 |
| Лабораторная работа № 2. Структура и процесс обработки программ на языке ассемблера NASM | 17 |
| 2.1. Цель работы | 17 |
| 2.2. Предварительные сведения | 17 |
| 2.3. Порядок выполнения работы | 23 |
| 2.4. Содержание отчёта | 24 |
| 2.5. Контрольные вопросы | 24 |
| Лабораторная работа № 3. Трансляция программ на языке ассемблера. Системные вызовы в ОС GNU Linux | 25 |
| 3.1. Цель работы | 25 |
| 3.2. Предварительные сведения | 25 |
| 3.3. Порядок выполнения работы | 33 |
| 3.4. Содержание отчёта | 33 |
| 3.5. Контрольные вопросы | 33 |
| Лабораторная работа № 4. Работа с файлом листинга | 35 |
| 4.1. Цель работы | 35 |
| 4.2. Предварительные сведения | 35 |
| 4.3. Порядок выполнения работы | 38 |
| 4.4. Содержание отчёта | 39 |
| 4.5. Контрольные вопросы | 39 |
| Лабораторная работа № 5. Отладчик GDB. Программирование цикла с переадресацией. Организация стека. | 40 |
| 5.1. Цель работы | 40 |
| 5.2. Предварительные сведения | 40 |
| 5.3. Порядок выполнения работы | 48 |
| 5.4. Содержание отчёта | 49 |
| 5.5. Контрольные вопросы | 49 |
| Лабораторная работа № 6. Отладчик EDB. Понятие подпрограммы | 50 |
| 6.1. Цель работы | 50 |
| 6.2. Предварительные сведения | 50 |
| 6.3. Порядок выполнения работы | 57 |
| 6.4. Содержание отчёта | 58 |

| | |
|--|---------------|
| 6.5. Контрольные вопросы | 58 |
| Лабораторная работа № 7. Побитовые операции | 60 |
| 7.1. Цель работы | 60 |
| 7.2. Предварительные сведения | 60 |
| 7.3. Порядок выполнения работы | 62 |
| 7.4. Содержание отчёта | 62 |
| 7.5. Контрольные вопросы | 63 |
| Учебно-методический комплекс | 65 |
| Программа дисциплины | 67 |
| Паспорт фонда оценочных средств | 77 |
| Фонд оценочных средств | 79 |
| Сведения об авторах | 87 |

Лабораторная работа № 1. Основы интерфейса командной строки ОС GNU Linux

1.1. Цель работы

Приобретение практических навыков общения с операционной системой на уровне командной строки (вход и выход, оперативная помощь, работа с буфером команд, организация файловой системы).

1.2. Предварительные сведения

1.2.1. Введение в GNU Linux

Операционная система (ОС) — это комплекс взаимосвязанных программ, предназначенных для управления ресурсами компьютера и организации взаимодействия с пользователем. Сегодня наиболее известными операционными системами являются ОС семейства Microsoft Windows и UNIX-подобные системы.

GNU Linux — семейство переносимых, многозадачных и многопользовательских операционных систем, на базе ядра Linux, включающих тот или иной набор утилит и программ проекта GNU, и, возможно, другие компоненты. Как и ядро Linux, системы на его основе, как правило, создаются и распространяются в соответствии с моделью разработки свободного и открытого программного обеспечения (Open-Source Software). Linux-системы распространяются в основном бесплатно в виде различных дистрибутивов.

Дистрибутив GNU Linux — общее определение ОС, использующих ядро Linux и набор библиотек и утилит, выпускаемых в рамках проекта GNU, а также графическую оконную подсистему X Window System. Дистрибутив готов для конечной установки на пользовательское оборудование. Кроме ядра и, собственно, операционной системы дистрибутивы обычно содержат широкий набор приложений, таких как редакторы документов и таблиц, мультимедийные проигрыватели, системы для работы с базами данных и т.д. Существуют дистрибутивы, разрабатываемые как при коммерческой поддержке (Red Hat / Fedora, SLED / OpenSUSE, Ubuntu), так и исключительно усилиями добровольцев (Debian, Slackware, Gentoo, ArchLinux).

1.2.2. Файловая структура GNU Linux: каталоги и файлы

Файловая система определяет способ организации, хранения и именования данных на носителях информации в компьютерах и представляет собой иерархическую структуру в виде вложенных друг в друга каталогов

(директорий), содержащих все файлы. В ОС Linux каталог, который является «вершиной» файловой системы, называется **корневым каталогом**, обозначается символом `/` и содержит все остальные каталоги и файлы.

В большинстве Linux-систем поддерживается стандарт иерархии файловой системы (Filesystem Hierarchy Standard, FHS), унифицирующий местонахождение файлов и каталогов. Это означает, что в корневом каталоге находятся только подкаталоги со стандартными именами и типами данных, которые могут попасть в тот или иной каталог. Так, в любой Linux-системе всегда есть каталоги `/etc`, `/home`, `/usr/bin` и т.п. В табл. 1.1 приведено краткое описание нескольких каталогов.

Таблица 1.1

Описание некоторых каталогов файловой системы GNU Linux

| Каталог | Описание |
|---------------------|---|
| <code>/</code> | Корневая директория, содержащая всю файловую иерархию |
| <code>/bin</code> | Основные системные утилиты, необходимые как в однопользовательском режиме, так и при обычной работе всем пользователям (например: <code>cat</code> , <code>ls</code> , <code>cp</code>) |
| <code>/etc</code> | Общесистемные конфигурационные файлы и файлы конфигурации установленных программ |
| <code>/home</code> | Содержит домашние директории пользователей, которые, в свою очередь, содержат персональные настройки и данные пользователя |
| <code>/media</code> | Точки монтирования для сменных носителей, таких как CD-ROM, DVD-ROM, flash |
| <code>/root</code> | Домашняя директория пользователя <code>root</code> |
| <code>/tmp</code> | Временные файлы |
| <code>/usr</code> | Вторичная иерархия для данных пользователя; содержит большинство пользовательских приложений и утилит, используемых в многопользовательском режиме; может быть смонтирована по сети только для чтения и быть общей для нескольких машин |

Обратиться к файлу, расположенному в каком-то каталоге, можно указав путь к нему. Существует несколько видов путей к файлу:

- **полный или абсолютный путь** — начинается от корня (`/`), образуется перечислением всех каталогов, разделённых прямым слешем (`/`), и завершается именем файла (например, полный путь к файлу `addition.txt`

из каталога `user` в каталоге `home`, находящемся в корневом каталоге, будет иметь вид: `/home/user/documents/addition.txt`;

- **относительный путь** — так же как и полный путь, строится перечислением через `/` всех каталогов, но начинается от текущего каталога (каталога, в котором «находится» пользователь), т.е. пользователь, находясь в каталоге `user`, может обратиться к файлу `addition.txt`, указав относительный путь `documents/addition.txt`.

Таким образом, в Linux если имя объекта начинается с `/`, то системой это интерпретируется как полный путь, в любом другом случае — как относительный.

В Linux любой пользователь имеет **домашний каталог**, который обычно расположен в каталоге `/home` и, как правило, имеет имя пользователя. В домашних каталогах хранятся документы и настройки пользователя. Для обозначения домашнего каталога используется знак тильды (`~`). При переходе из домашнего каталога знак тильды будет заменён на имя нового текущего каталога.

1.2.3. Введение в командную строку GNU Linux

Работу ОС GNU Linux можно представить в виде функционирования множества взаимосвязанных процессов. При загрузке системы сначала запускается ядро, которое, в свою очередь, запускает оболочку ОС (от англ. shell «оболочка»). Взаимодействие пользователя с системой Linux (работа с данными и управление работающими в системе процессами) происходит в интерактивном режиме посредством командного языка. Оболочка операционной системы (или командная оболочка, интерпретатор команд) — интерпретирует (т.е. переводит на машинный язык) вводимые пользователем команды, запускает соответствующие программы (процессы), формирует и выводит ответные сообщения. Кроме того, на языке командной оболочки можно писать небольшие программы для выполнения ряда последовательных операций с файлами и содержащимися в них данными — сценарии (скрипты).

В операционные системы Windows включён командный интерпретатор `cmd.exe`, в MS-DOS — `command.com`. В GNU Linux у пользователя есть возможность менять командный интерпретатор, используемый по умолчанию. Из командных оболочек GNU Linux наиболее популярны `bash`, `csh`, `ksh`, `zsh`. В качестве предустановленной командной оболочки GNU Linux используется одна из наиболее распространённых разновидностей командной оболочки — `bash` (Bourne again shell).

В GNU Linux доступ пользователя к командной оболочке обеспечивается через терминал (или консоль). Запуск терминала можно осуществить через главное меню Приложения Стандартные Терминал (или Консоль) или нажав Ctrl + Alt + t.

Интерфейс командной оболочки очень прост. Обычно он состоит из приглашения командной строки (строки, оканчивающейся символом `$`), по которому пользователь вводит команды:

```
iivanova@dk4n31:~$
```

Это приглашение командной оболочки, которое несёт в себе информацию об имени пользователя (`iivanova`), имени компьютера (`dk4n31`) и текущем каталоге, в котором находится пользователь, в данном случае это домашний каталог пользователя, обозначенный как (`~`).

Команды могут быть использованы с ключами (или опциями) — указаниями, модифицирующими поведение команды. Ключи обычно начинаются с символа '-' или '--' и часто состоят из одной буквы. Кроме ключей после команды могут быть использованы аргументы (параметры) — названия объектов, для которых нужно выполнить команду (например, имена файлов и каталогов).

Ввод команды завершается нажатием клавиши `Enter`, после чего команда передаётся оболочке на исполнение. Результатом выполнения команды могут являться сообщения о ходе выполнения команды или об ошибках. Появление приглашения командной строки говорит о том, что выполнение команды завершено.

Иногда в GNU Linux имена программ и команд слишком длинные, однако `bash` может завершать имена при их вводе в терминале. Нажав клавишу `Tab`, можно завершить имя команды, программы или каталога. Например, предположим, что нужно использовать программу `mcedit`. Для этого наберите в командной строке `mc`, затем нажмите один раз клавишу `Tab`. Если ничего не происходит, то это означает, что существует несколько возможных вариантов завершения команды. Нажав клавишу `Tab` ещё раз, можно получить список имён, начинающихся с `mc`:

```
user@dk4n31:~$ mc
mc      mcd      mcedit    mclasserase  mcookie    mcview
mcat    mcdiff    mcheck    mcomp        mcopy
user@dk4n31:~$ mc
```

Более подробно о работе в операционной системе Linux см., например, в [2; 3].

1.2.4. Базовые команды `bash`

Первые задачи, которые приходится решать в любой системе, — работа с данными (обычно хранящимися в файлах) и управление работающими в системе программами (процессами). Для получения достаточно подробной информации по каждой из команд используйте команду `man`, например:

```
user@dk4n31:~$ man ls
```

1.2.4.1. Команда `pwd`: вывод имени текущего каталога

Определить каталог, в котором в данный момент находится пользователь, можно набрав команду `pwd` (Print Working Directory):


```
user@dk4n31:/usr/src$ pwd
/usr/src
user@dk4n31:/usr/src$
```

1.2.4.2. Команда **cd**: смена каталога

Команда **cd** (Change Directory) позволяет сменить текущий каталог на другой, указав путь к нему в качестве параметра. Команда **cd** работает как с абсолютными, так и с относительными путями.

Предположим, что вы находитесь в своём домашнем каталоге и хотите перейти в каталог **tmp**, расположенный также в домашнем каталоге. Для этого достаточно ввести относительный путь:

```
user@dk4n31:~$ cd tmp
user@dk4n31:~/tmp$
```

Но если вы хотите перейти в каталог **tmp**, расположенный в корне системы, то нужно ввести абсолютный путь к нему:

```
user@dk4n31:~$ cd /tmp
user@dk4n31:/tmp$
```

Ещё один пример использования абсолютного пути для перехода в каталог **/usr/bin**:

```
user@dk4n31:~$ cd /usr/bin
user@dk4n31:/usr/bin$
```

Запуск команды **cd** без параметров обеспечивает переход в домашний каталог пользователя:

```
user@dk4n31:/usr/bin$ cd
user@dk4n31:~$
```

Можно использовать комбинацию **cd ..** для перехода на один каталог выше по иерархии, а сочетание **cd -** вернёт пользователя в последний посещённый им каталог.

1.2.4.3. Команда **ls**: вывод списка файлов

Команда **ls** (LiSt) выдаёт список файлов указанного каталога и имеет следующий синтаксис:

```
ls [опции] [файл|каталог] [файл|каталог...]
```

Если в командной строке в качестве параметров не указано имя каталога, то команда **ls** выведет список файлов текущего каталога. Для данной команды существует довольно много опций, ниже дано описание некоторых из них:

- a: вывод списка всех файлов, включая скрытые файлы (в Linux названия скрытых файлов начинаются с точки);
- R: рекурсивный вывод списка файлов и подкаталогов;
- h: вывод для каждого файла его размера;
- l: вывод дополнительной информации о файлах (права доступа, владельцы и группы, размеры файлов и время последнего доступа);

- i: вывод уникального номера файла (inode) в файловой системе перед каждым файлом;
- d: обработка каталогов, указанных в командной строке, так, как если бы они были обычными файлами, вместо вывода списка их файлов.

Примеры:

- команда `ls -R` рекурсивно выводит список содержимого текущего каталога;
- команда `ls -ls images/ ..` выводит список файлов каталога `images` и родительского по отношению к текущему каталога, при этом для каждого файла указан номер inode и его размер в килобайтах;
- команда `ls -l images/*.png` выводит список всех файлов в каталоге `images`, чьи имена заканчиваются на `.png`, включая скрытый файл `.png`, если таковой существует.

1.2.4.4. Команды `mkdir`, `touch`: создание пустых каталогов и файлов

Для создания каталогов используется команда `mkdir` (MaKe DiRectory — создать каталог). Её синтаксис имеет вид:

```
mkdir [опции] <каталог> [каталог...]
```

Команда `mkdir` имеет несколько опций, например, использование опции `-p` позволит выполнить два действия:

- 1) создать родительские каталоги, если они не существуют (без этой опции `mkdir` выдаст ошибку об отсутствии заявленных каталогов);
- 2) завершит работу без ошибки, если каталог, который необходимо создать, уже существует (без указания этой опции команда `mkdir` выдаст сообщение об ошибке, сообщая, что каталог уже существует).

Приведём несколько примеров.

Создать каталог `image` в текущем каталоге:

```
mkdir image
```

Создать каталог `misc` в каталоге `images` и каталог `docs` в текущем каталоге:

```
mkdir -p images/misc docs
```

Для создания файлов может быть использована команда `touch`, которая имеет следующий синтаксис:

```
touch [опции] файл [файл...]
```

Например, для создания в текущем каталоге файла с именем `file1` и в существующем подкаталоге `images` текущего каталога файла с именем `file2` следует ввести команду:

```
touch file1 images/file2
```

1.2.4.5. Команда `rm`: удаление файлов или каталогов

Команда `rm` (ReMove — удалить) удаляет файлы и (или) каталоги и имеет следующий синтаксис:

```
rm [опции] <файл|каталог> [файл|каталог...]
```

Опции команды `rm`:

- r или -R: рекурсивное удаление (это обязательная опция для удаления любого каталога, пустого или содержащего файлы и (или) подкаталоги);
- i: запрос подтверждения перед удалением;
- v: вывод подробной информации при выполнении команды.

Для удаления пустых каталогов можно воспользоваться командой `rmdir`.

Приведём несколько примеров.

Запросив подтверждение на удаление каждого файла в текущем каталоге, удалить в подкаталоге `images` все файлы с именами, заканчивающимися на `.jpg`, а также `file1`:

```
rm -i images/*.jpg file1
```

Рекурсивно удалить из текущего каталога без запроса подтверждения на удаление каталог `misc` в подкаталоге `images`, а также файлы, чьи имена начинаются с `file`:

```
rm -R images/misc/ file*
```

Команда `rm` удаляет файлы безвозвратно, и не существует способа для их восстановления.

1.2.4.6. Команда `mv`: перемещение или удаление файлов

Команда `mv` (MoVe — переместить) служит для перемещения файлов и каталогов и имеет следующий синтаксис:

```
mv [опции] <файл|каталог> [файл|каталог...] <назначение>
```

При перемещении нескольких файлов местом назначения должен быть каталог.

Некоторые опции:

- f: принудительное выполнение операции (предупреждение не будет выводиться даже при перезаписи существующего файла);
- i: запрашивается подтверждение перед перезаписью существующего файла;
- v: подробный режим, который сообщает обо всех изменениях и действиях при выполнении команды.

Приведём несколько примеров.

Переместить все файлы, чьи имена заканчиваются на `.png`, из каталога `~/tmp/pics/` в текущий каталог (`.`), запрашивая подтверждение перед перезаписью:

```
mv -i ~/tmp/pics/*.png .
```

Переименовать файл или каталог `foo` в `bar`; если существует каталог `bar`, то произойдёт перемещение файла `foo` или всего каталога (с файлами и подкаталогами, содержащимися в нём) в каталог `bar`:

```
mv foo bar
```

Переместить без запроса подтверждения все файлы из текущего каталога с именами, начинающимися с `file`, и весь каталог `images` в каталог `trash`, показывая порядок выполнения всех операций:

```
mv -vf file* images/ trash/
```

1.2.4.7. Команда `cp`: копирование файлов и каталогов

Команда `cp` (CoPy — копировать) копирует файлы и каталоги и имеет следующий синтаксис:

```
cp [опции] <файл|каталог> [файл|каталог ...] <назначение>
```

Некоторые опции команды `cp`:

- R: рекурсивное копирование; является обязательной опцией для копирования каталогов;
- i: запрос подтверждения перед перезаписью любых файлов;
- f: заменяет любые существующие файлы без запроса подтверждения;
- v: подробный режим, сообщает обо всех изменениях и действиях.

Приведём несколько примеров.

Запросить подтверждение и скопировать все файлы из каталога `/image` в каталог `images` в текущем каталоге:

```
cp -i /image/* images/
```

Скопировать весь каталог `images` и все файлы из каталога `/docs/text/` в каталог `mytext`:

```
cp -vR images/ /docs/text/* mytext/
```

Сделать копию файла `addition` в файл с именем `subtraction` в текущем каталоге:

```
cp addition subtraction
```

1.2.4.8. Команда `cat`: вывод содержимого файлов

Команда `cp` объединяет файлы и выводит их на стандартный вывод (обычно это экран):

```
$ cat /etc/hosts
```

```
#  
# /etc/hosts: static lookup table for host names  
#  
  
#<ip-address>      <hostname.domain.org>    <hostname>  
127.0.0.1          localhost.localdomain    localhost  
  
# End of file
```

Более подробно о работе в `bash` см. в [1; 4–6].

1.2.5. Основные возможности текстового редактора `mcedit`

`mcedit` — является встроенным текстовым редактором для файлового менеджера Midnight Commander (или `mc`), который позволяет редактировать различные файлы размером до 64 Мб. К основным возможностям этого редактора можно отнести: копирование блока символов, перемещение, удаление, вырезка, вставка и др.

Например, чтобы создать в текущем каталоге файл `addition.txt` и начать его редактирование, можно набрать:

```
mcedit addition.txt
```

В общем случае для запуска `mcedit` может быть использован следующий синтаксис:

```
mcedit [-bcCdfhstVx?] [+число] file
```

Некоторые параметры `mcedit` пояснены в табл. 1.2.

Таблица 1.2

Некоторые параметры `mcedit`

| Параметр | Действие |
|----------|--|
| +число | переход к указанной числом строке |
| -b | черно-белая цветовая гамма |
| -c | цветовой режим ANSI для терминалов без поддержки цвета |
| -d | отключить поддержку мыши |
| -V | вывести версию программы |

Клавиша `F9` служит для вызова меню. В табл. 1.3 приведён список наиболее часто используемых горячих клавиш (`Meta` — условное обозначение для набора мета-клавиш, обычно это `Alt` или `Esc`):

Таблица 1.3

Список наиболее часто используемых горячих клавиш `mcedit`

| Горячие клавиши | Действие |
|---------------------------------------|--|
| <code>Ctrl</code> + <code>Ins</code> | копировать |
| <code>Shift</code> + <code>Ins</code> | вставить |
| <code>Shift</code> + <code>Del</code> | вырезать |
| <code>Ctrl</code> + <code>Del</code> | удалить выделенный текст |
| <code>F3</code> | начать выделение текста (повторное нажатие <code>F3</code> закончит выделение) |
| <code>Shift</code> + <code>F3</code> | начать выделение блока текста (повторное нажатие <code>F3</code> закончит выделение) |
| <code>F5</code> | скопировать выделенный текст |
| <code>F6</code> | переместить выделенный текст |
| <code>F8</code> | удалить выделенный текст |
| <code>Meta</code> + <code>l</code> | переход к строке по её номеру |
| <code>Meta</code> + <code>q</code> | вставка литерала (непечатного символа) |

Таблица 1.3

Список наиболее часто используемых горячих клавиш `mcedit` (окончание)

| Горячие клавиши | Действие |
|---|---|
| <code>Meta</code> + <code>t</code> | сортировка строк выделенного текста |
| <code>Meta</code> + <code>u</code> | выполнить внешнюю команду и вставить в позицию под курсором её вывод |
| <code>Ctrl</code> + <code>f</code> | занести выделенный фрагмент во внутренний буфер обмена <code>mc</code> (записать во внешний файл) |
| <code>Ctrl</code> + <code>k</code> | удалить часть строки до конца строки |
| <code>Ctrl</code> + <code>n</code> | создать новый файл |
| <code>Ctrl</code> + <code>s</code> | включить или выключить подсветку синтаксиса |
| <code>Ctrl</code> + <code>t</code> | выбрать кодировку текста |
| <code>Ctrl</code> + <code>u</code> | отменить действия |
| <code>Ctrl</code> + <code>x</code> | перейти в конец следующего слова |
| <code>Ctrl</code> + <code>y</code> | удалить строку |
| <code>Ctrl</code> + <code>z</code> | перейти на начало предыдущего слова |
| <code>Shift</code> + <code>F5</code> | вставка текста из внутреннего буфера обмена <code>mc</code> (прочитать внешний файл) |
| <code>Meta</code> + <code>Enter</code> | диалог перехода к определению функции |
| <code>Meta</code> + <code>-</code> | возврат после перехода к определению функции |
| <code>Meta</code> + <code>+</code> | переход вперёд к определению функции |
| <code>Meta</code> + <code>n</code> | включение/отключение отображения номеров строк |
| <code>Tab</code> | отодвигает вправо выделенный текст, если выключена опция «Постоянные блоки» |
| <code>Meta</code> + <code>Tab</code> | отодвигает влево выделенный текст, если выключена опция «Постоянные блоки» |
| <code>Shift</code> + <code>Стрелки</code> | выделение текста |
| <code>Meta</code> + <code>Стрелки</code> | выделение вертикального блока |
| <code>Meta</code> + <code>Shift</code> + <code>-</code> | переключение режима отображения табуляций и пробелов |
| <code>Meta</code> + <code>Shift</code> + <code>+</code> | переключение режима «Автовывравнивание возврата каретки» |





Клавиши (обычно `Alt` + `Tab` или `Esc` + `Tab`) служат для автозавершения и завершают слово, на котором находится курсор, используя ранее использовавшиеся в файле.






При помощи команды `man mc` можно получить дополнительную информацию по `mc`.




1.2.6. Полезные комбинации клавиш

Для удобства и экономии времени при работе в терминале существует большое количество сокращённых клавиатурных команд.

Клавиши со стрелками.

Клавиши  и  позволяют увидеть историю предыдущих команд в `bash`. Количество хранимых строк определено в переменной окружения `HISTSIZE`. Клавиши  и  перемещают курсор влево и вправо в текущей строке, позволяя редактировать команды.

Сочетания клавиш  и  перемещают курсор в начало и в конец текущей строки. Клавиши  удаляет всё от текущей позиции курсора до конца строки, а  или  удаляют слово перед курсором.

Сочетание клавиш  в пустой строке служит для завершения текущего сеанса. Для завершения выполняющейся в данный момент команды можно использовать . Также данное сочетание отменит редактирование командной строки и вернёт приглашение командной строки.  очищает экран.

1.3. Порядок выполнения работы

1. Войдите в систему ОС Linux, введя свои логин и пароль.
2. Последовательно изучите основные консольные команды, описанные в указаниях к работе.
3. Воспользовавшись командой `pwd`, узнайте полный путь к своей домашней директории.
4. Пользуясь командами `cd` и `ls`, посмотрите содержимое корневого каталога, домашнего каталога, каталога `/etc` и каталога `/usr/local`.
5. Пользуясь изученными консольными командами, создайте в своём домашнем каталоге подкаталог `lab01`, а в нём файл `addition.txt`.
6. С помощью стандартного текстового редактора (например, редактора `mcedit`) запишите в файл `addition.txt` свои имя и фамилию.
7. Пользуясь командой `ls`, убедитесь, что все действия выполнены успешно.
8. Выведите на экран содержимое файла `addition.txt`, используя команду `cat`.
9. Пользуясь изученными консольными командами, удалите созданные файл `addition.txt` и каталог `~/lab01`.

1.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;

- 3) описание задания и результаты его выполнения:
 - листинги программ (если они есть в задании);
 - краткое описание всех действий, сделанных при выполнении заданий;
 - результаты выполнения команд и программ (снимок экрана).
- 4) выводы, согласованные с целью работы;
- 5) ответы на контрольные вопросы.

1.5. Контрольные вопросы

1. Дайте определение командной строки. Приведите примеры.
2. Как определить абсолютный путь к текущей директории?
3. Как удалить файл и каталог?
4. Как определить, какие команды выполнил пользователь в сеансе работы?
5. Какая информация выводится на экран о файлах и каталогах, если используется опция `-l` в команде `ls`?
6. Что такое относительный путь к файлу? Приведите примеры.
7. Назовите необходимые условия, для того чтобы пользователь мог начать работать в системе типа UNIX (на примере терминального класса, в котором выполнялась лабораторная работа).

Список литературы

1. *Newham C.* Learning the bash Shell: Unix Shell Programming. — O'Reilly Media, 2005. — 354 p. — (In a Nutshell). — ISBN 0596009658.
2. *Робачевский А., Немнюгин С., Стесик О.* Операционная система UNIX. — 2-е изд. — БХВ-Петербург, 2010. — 656 с. — ISBN 978-5-94157-538-1.
3. *Таненбаум Э., Бос Х.* Современные операционные системы. — 4-е изд. — СПб. : Питер, 2015. — 1120 с. — (Классика Computer Science).
4. GNU Bash Manual. — 2016. — URL: <https://www.gnu.org/software/bash/manual/>.
5. *Robbins A.* Bash Pocket Reference. — O'Reilly Media, 2016. — 156 p. — ISBN 978-1491941591.
6. *Zarrelli G.* Mastering Bash. — Packt Publishing, 2017. — 502 p. — ISBN 9781784396879.

Лабораторная работа № 2. Структура и процесс обработки программ на языке ассемблера NASM

2.1. Цель работы

Освоение процедуры компиляции и сборки программ, написанных на ассемблере NASM.

2.2. Предварительные сведения

2.2.1. Основные принципы работы компьютера

Основными функциональными элементами любой электронно-вычислительной машины (ЭВМ) являются центральный процессор, память и периферийные устройства (рис. 2.1).

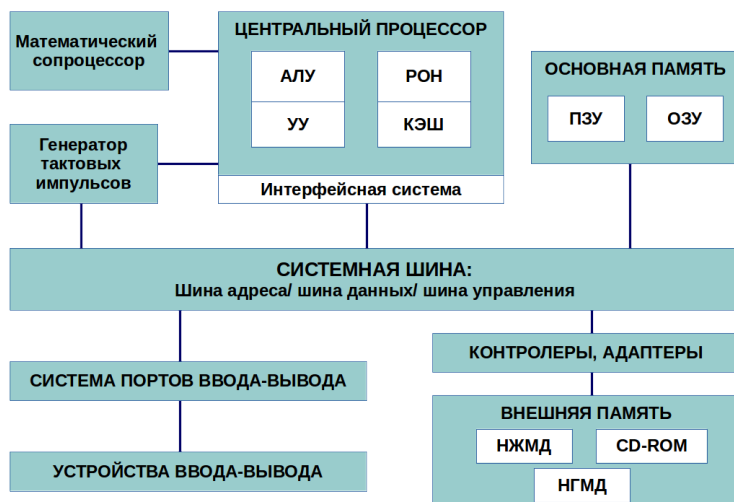


Рис. 2.1. Структурная схема ЭВМ

Взаимодействие этих устройств осуществляется через общую шину, к которой они подключены. Физически шина представляет собой большое количество проводников, соединяющих устройства друг с другом. В современных компьютерах проводники выполнены в виде электропроводящих дорожек на материнской (системной) плате.

Основной задачей процессора является обработка информации, а также организация координации всех узлов компьютера. В состав *центрального процессора (ЦП)* входят следующие устройства:

- *арифметико-логическое устройство (АЛУ)* — выполняет логические и арифметические действия, необходимые для обработки информации, хранящейся в памяти;
- *устройство управления (УУ)* — обеспечивает управление и контроль всех устройств компьютера;
- *регистры* — сверхбыстрая оперативная память небольшого объёма, входящая в состав процессора, для временного хранения промежуточных результатов выполнения инструкций; регистры процессора делятся на два типа: *регистры общего назначения* и *специальные регистры*.

Другим важным узлом ЭВМ является *оперативное запоминающее устройство (ОЗУ)*. ОЗУ — это быстродействующее энергозависимое запоминающее устройство, которое напрямую взаимодействует с узлами процессора, предназначенное для хранения программ и данных, с которыми процессор непосредственно работает в текущий момент. ОЗУ состоит из одинаковых пронумерованных ячеек памяти. Номер ячейки памяти — это адрес хранящихся в ней данных.

В состав ЭВМ также входят периферийные устройства, которые можно разделить на:

- *устройства внешней памяти*, которые предназначены для долговременного хранения больших объёмов данных (жёсткие диски, твердотельные накопители, магнитные ленты);
- *устройства ввода-вывода*, которые обеспечивают взаимодействие ЦП с внешней средой.

В основе вычислительного процесса ЭВМ лежит *принцип программного управления*. Это означает, что компьютер решает поставленную задачу как последовательность действий, записанных в виде программы. Программа состоит из машинных команд, которые указывают, какие операции и над какими данными (или операндами), в какой последовательности необходимо выполнить.

Набор машинных команд определяется устройством конкретного процессора. Коды команд представляют собой многоразрядные двоичные комбинации из 0 и 1. В коде машинной команды можно выделить две части: операционную и адресную. В операционной части хранится код команды, которую необходимо выполнить. В адресной части хранятся данные или адреса данных, которые участвуют в выполнении данной операции.

При выполнении каждой команды процессор выполняет определённую последовательность стандартных действий, которая называется *командным циклом процессора*. В самом общем виде он заключается в следующем:

- 1) формирование адреса в памяти очередной команды;

- 2) считывание кода команды из памяти и её дешифрация;
- 3) выполнение команды.

Данный алгоритм позволяет выполнить хранящуюся в ОЗУ программу. Кроме того, в зависимости от команды при её выполнении могут проходить не все этапы.

Более подробно введение о теоретических основах архитектуры ЭВМ см. в [3; 6].

2.2.2. Ассемблер и язык ассемблера

Язык ассемблера (assembly language, сокращённо asm) — машинно-ориентированный язык низкого уровня. Можно считать, что он больше любых других языков приближен к архитектуре ЭВМ и её аппаратным возможностям, что позволяет получить к ним более полный доступ, нежели в языках высокого уровня, таких как C/C++, Perl, Python и пр. Заметим, что получить полный доступ к ресурсам компьютера в современных архитектурах нельзя, самым низким уровнем работы прикладной программы является обращение напрямую к ядру операционной системы. Именно на этом уровне и работают программы, написанные на ассемблере. Но в отличие от языков высокого уровня ассемблерная программа содержит только тот код, который ввёл программист. Таким образом язык ассемблера — это язык, с помощью которого понятным для человека образом пишутся команды для процессора.

Следует отметить, что процессор понимает не команды ассемблера, а последовательности из нулей и единиц — *машинные коды*. До появления языков ассемблера программистам приходилось писать программы, используя только лишь машинные коды, которые были крайне сложны для запоминания, так как представляли собой числа, записанные в двоичной или шестнадцатеричной системе счисления. Преобразование или *трансляция* команд с языка ассемблера в исполняемый машинный код осуществляется специальной программой транслятором — *Ассемблер*.

Программы, написанные на языке ассемблера, не уступают в качестве и скорости программам, написанным на машинном языке, так как транслятор просто переводит мнемонические обозначения команд в последовательности бит (нулей и единиц).

Используемые мнемоники обычно одинаковы для всех процессоров одной архитектуры или семейства архитектур (среди широко известных — мнемоники процессоров и контроллеров x86, ARM, SPARC, PowerPC, M68k). Таким образом для каждой архитектуры существует свой ассемблер и, соответственно, свой язык ассемблера.

Наиболее распространёнными ассемблерами для архитектуры x86 являются:

- для DOS/Windows: Borland Turbo Assembler (TASM), Microsoft Macro Assembler (MASM) и Watcom Assembler (WASM);
- для GNU/Linux: gas (GNU Assembler), использующий AT&T-синтаксис, в отличие от большинства других популярных ассемблеров, которые используют Intel-синтаксис.

Более подробно о языке ассемблера см., например, в [5].

В нашем курсе будет использоваться ассемблер NASM (Netwide Assembler) [1; 2; 4].

NASM — это открытый проект ассемблера, версии которого доступны под различные операционные системы и который позволяет получать объектные файлы для этих систем. В NASM используется Intel-синтаксис и поддерживаются инструкции x86-64.

Типичный формат записи команд NASM имеет вид:

```
[метка:]  мнемокод [операнд {, операнд}]  [; комментарий]
```

Здесь *мнемокод* — непосредственно мнемоника инструкции процессору, которая является обязательной частью команды. Операндами могут быть числа, данные, адреса регистров или адреса оперативной памяти.

Программа на языке ассемблера также может содержать *директивы* — инструкции, не переводящиеся непосредственно в машинные команды, а управляющие работой транслятора. Например, директивы используются для определения данных (констант и переменных) и обычно пишутся большими буквами.

2.2.2.1. Структура ассемблерной программы

Рассмотрим пример простой программы на языке ассемблера NASM. Традиционно первая программа выводит приветственное сообщение Hello world! (Здравствуй мир!) на экран. Назовём файл с кодом программы hello.asm и запишем в него следующий текст:

```
; hello.asm
SECTION .data                ; Начало секции данных
    hello:    DB 'Hello world!',10 ; 'Hello world!' плюс
                                ; символ возврата каретки
    helloLen: EQU $-hello      ; Длина строки 'Hello world!'

SECTION .text                ; Начало секции кода
    GLOBAL _start

_start:                      ; Точка входа в программу
    mov eax,4                ; Системный вызов для записи (sys_write)
    mov ebx,1                ; Описатель файла $1$ - стандартный вывод
    mov ecx,hello            ; Адрес строки hello в ecx
    mov edx,helloLen         ; Размер строки hello

    int 80h                  ; Вызов ядра

    mov eax,1                ; Системный вызов для выхода (sys_exit)
    mov ebx,0                ; Выход с кодом возврата $0$ (без ошибок)
    int 80h                  ; Вызов ядра
```

В отличие от многих современных высокоуровневых языков программирования, в ассемблерной программе каждая команда располагается

на *отдельной строке*. Размещение нескольких команд на одной строке недопустимо. Синтаксис ассемблера NASM является *чувствительным к регистру*, т. е. есть разница между большими и малыми буквами.

2.2.2.2. Процесс создания и обработки программы на языке ассемблера

Из-за специфики программирования, а также по традиции для создания программ на языке ассемблера обычно пользуются утилитами командной строки (хотя поддержка ассемблера есть в некоторых универсальных интегрированных средах).

Процесс создания ассемблерной программы можно изобразить в виде следующей схемы (см. рис. 2.2).

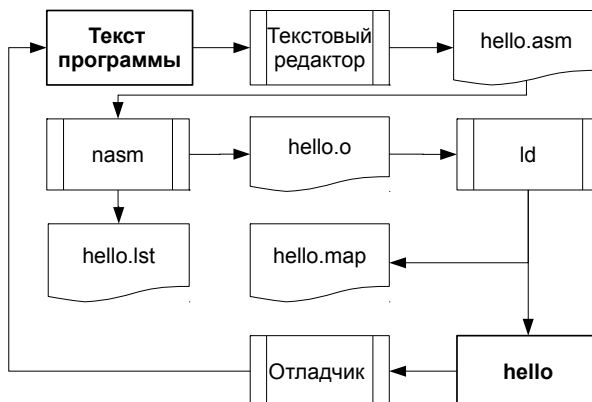


Рис. 2.2. Процесс создания ассемблерной программы

В процессе создания ассемблерной программы можно выделить четыре шага:

1. **Набор текста** программы в текстовом редакторе и сохранение её в отдельном файле. Каждый файл имеет свой тип (или расширение), который определяет назначение файла. Файлы с исходным текстом программ на языке ассемблера имеют тип *asm*.
2. **Трансляция** — преобразование с помощью транслятора, например *nasm*, текста программы в машинный код, называемый *объектным*. На данном этапе также может быть получен листинг программы, содержащий кроме текста программы различную дополнительную информацию, созданную транслятором. Тип объектного файла — *o*, файла листинга — *lst*.
3. **Компоновка или линковка** — этап обработки объектного кода компоновщиком (*ld*), который принимает на вход объектные файлы и собирает

по ним исполняемый файл. Исполняемый файл обычно не имеет расширения. Кроме того, можно получить файл карты загрузки программы в ОЗУ, имеющий расширение `map`.

4. **Запуск программы.** Конечной целью является работоспособный исполняемый файл. Ошибки на предыдущих этапах могут привести к некорректной работе программы, поэтому может присутствовать этап *отладки* программы при помощи специальной программы — отладчика. При нахождении ошибки необходимо провести коррекцию программы, начиная с первого шага.

2.2.3. Транслятор NASM

NASM превращает текст программы в объектный код. Например, для компиляции приведённого выше текста программы «Hello World» необходимо написать:

```
nasm -f elf64 hello.asm
```

Если текст программы набран без ошибок, то транслятор преобразует текст программы из файла `hello.asm` в объектный код, который запишется в файл `hello.o`. Таким образом, имена всех файлов получаются из имени входного файла и расширения по умолчанию. При наличии ошибок объектный файл не создаётся, а после запуска транслятора появятся сообщения об ошибках или предупреждения.

NASM не запускают без параметров. Ключ `-f` указывает транслятору, что требуется создать бинарные файлы в формате ELF. Следует отметить, что формат `elf64` позволяет создавать исполняемый код, работающий под 64-битными версиями Linux. Для 32-битных версий ОС указываем в качестве формата просто `elf`.

NASM всегда создаёт выходные файлы в *текущем* каталоге.

2.2.3.1. Расширенный синтаксис командной строки NASM

Полный вариант командной строки `nasm` выглядит следующим образом:

```
nasm [-@ косвенный_файл_настроек] [-o объектный_файл] [-f  
→ формат_объектного_файла] [-l листинг] [параметры...]  
→ [--] исходный_файл
```

Для более подробной информации см. `man nasm`. Для получения списка форматов объектного файла см. `nasm -hf`.

Приведём пример. Команда `nasm -f elf64 -g -l main.lst main.asm` скомпилирует исходный файл `main.asm` в `main.o`, при этом формат выходного файла будет `elf64`, и в него будут включены символы для отладки (опция `-g`), кроме того, будет создан файл листинга `main.lst`.

2.2.3.2. Компоновщик LD

Как видно из схемы на рис. 2.2, чтобы получить исполняемую программу, объектный файл необходимо передать на обработку компоновщику:

```
ld -o hello hello.o
```

Ключ `-o` с последующим значением задаёт в данном случае имя создаваемого исполняемого файла.

В общем виде компоновщик `ld` имеет следующий формат командной строки:

```
ld [параметры] объектные_файлы...
```

Например, команда `ld -Map main.map -o main main.o` создаст исполняемый файл `main` из объектного файла `main.o`, при этом создавая карту памяти в файл `main.map`.

Формат командной строки LD можно увидеть, набрав `ld --help`. Для получения более подробной информации см. `man ld`.

Компоновщик `ld` не предполагает по умолчанию расширений для файлов, но принято использовать следующие расширения:

- `o` для объектных файлов;
- *без расширения* для исполняемых файлов;
- `map` для файлов схемы программы;
- `lib` для библиотек.

Если имя выполняемого файла не определено, оно будет образовано из имени первого объектного файла.

2.2.3.3. Запуск исполняемого файла

Запустить на выполнение созданный исполняемый файл, находящийся в текущем каталоге, можно, набрав в командной строке:

```
./hello
```

2.3. Порядок выполнения работы

1. Создайте в своём домашнем каталоге новый подкаталог с именем `lab02` и файл `lab2.asm`:

```
cd  
mkdir lab02  
touch lab2.asm
```

С помощью редактора `mcedit` введите в файл `lab2.asm` текст программы `hello.asm` из п. 2.2.2.1, пользуясь правилами оформления ассемблерных программ (не копируйте текст программы из файла `pdf`, так как некоторые символы могут скопироваться некорректно и программа не будет корректно транслироваться).

2. Оттранслируйте полученный текст программы в объектный файл.
3. Выполните компоновку объектного файла и запустите получившийся исполняемый файл.

4. Измените в тексте программы выводимую на экран строку с `Hello world!` на свою фамилию. Повторите пункты 2 и 3.

2.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;
- 3) описание процесса выполнения задания. Для каждого действия, производимого в командной строке, в отчёт следует включить:
 - краткое описание действия;
 - вводимую команду или команды;
 - результаты выполнения команд и программ (снимок экрана);
- 4) выводы, согласованные с целью работы.
- 5) ответы на контрольные вопросы.

2.5. Контрольные вопросы

1. Какие основные отличия ассемблерных программ от программ на языках высокого уровня?
2. В чём состоит отличие инструкции от директивы на языке ассемблера?
3. Перечислите основные правила оформления программ на языке ассемблера.
4. Каковы этапы получения исполняемого файла?
5. Каково назначение этапа трансляции?
6. Каково назначение этапа компоновки?
7. Какие файлы могут создаваться при трансляции программы, какие из них создаются по умолчанию?
8. Каковы форматы файлов для `nasm` и `ld`?

Список литературы

1. Расширенный ассемблер: `NASM`. — 2001. — URL: <https://www.opennet.ru/docs/RUS/nasm/>.
2. *Столяров А.* Программирование на языке ассемблера `NASM` для ОС `Unix`. — 2-е изд. — М. : МАКС Пресс, 2011.
3. *Новожилов О. П.* Архитектура ЭВМ и систем. — М. : Юрайт, 2016.
4. The `NASM` documentation. — 2017. — URL: <https://www.nasm.us/docs.php>.
5. *Куляс О. Л., Никитин К. А.* Курс программирования на `ASSEMBLER`. — М. : Солон-Пресс, 2017.
6. *Колдаев В. Д., Лутин С. А.* Архитектура ЭВМ. — М. : Форум, 2018.

Лабораторная работа № 3. Трансляция программ на языке ассемблера. Системные вызовы в ОС GNU Linux

3.1. Цель работы

Приобретение практических навыков по разработке небольших командных файлов. Освоение инструкций языка ассемблера `mov` и `int`.

3.2. Предварительные сведения

3.2.1. Трансляция и компоновка программ с помощью `make`

Программа `make` — это утилита, предназначенная для автоматизации действий по преобразованию файлов. В частности, с её помощью можно автоматизировать трансляцию и компоновку программ, написанных на ассемблере.

Как вы уже знаете из предыдущей лабораторной работы, чтобы преобразовать файл с исходным кодом на ассемблере (`main.asm`) в исполняемый файл (`main`), следует выполнить следующие две команды:

```
nasm -f elf64 -g -l main.lst main.asm
ld -Map main.map -o main main.o
```

В результате будет создан исполняемый файл `main`, а также объектный файл `main.o`, файл листинга `main.lst` и карта памяти `main.map`. Все дополнительные файлы можно удалить также из командной строки, используя команду `rm`:

```
rm main main.o main.lst main.map
```

В процессе разработки программного кода подобные команды вам придётся выполнять многократно. Кроме того, если программа будет состоять не из одного исходного файла, а из нескольких, то для каждого из них также придётся выполнять подобные команды. Поэтому удобно автоматизировать данные действия. Для этого и служит утилита `make`.

Вызов утилиты `make` происходит из командной строки посредством ввода одноимённой команды и нажатия клавиши `Enter`. Утилита `make` попытается найти в текущей директории файл с именем `Makefile` и выполнить правила преобразования, записанные в нём. Если `Makefile` с инструкциями отсутствует, то утилита завершит своё выполнение и выведет сообщение об ошибке.

`Makefile` представляет собой текстовый файл, который может содержать:

- комментарии;

- правила;
- макроопределения.

Создадим Makefile для автоматизации трансляции и линковки NASM программы. Для этого откроем текстовый редактор и наберём следующий тест (обратите внимание на наличие табуляции):

```
build: main.asm
    nasm -f elf64 -g -l main.lst main.asm
    ld -Map main.map -o main main.o
clean:
    rm -f main main.o main.lst main.map
    # здесь могут быть дополнительные команды
```

В примере созданы два правила с именами («целями») `build` и `clean`. Первое из них (`build`) выполняет трансляцию и сборку программы `main.asm`, а второе (`clean`) удаляет все созданные предыдущим правилом файлы. Таким образом, если набрать в командной строке `make build`, то будет создан исполняемый файл из `main.asm`. Если набрать `make clean`, то проект будет очищен, т.е. будут удалены все файлы, создаваемые `nasm` и `ld`.

Обратите внимание, что после цели `build` через двоеточие написано имя файла `main.asm`. Это указывает на то, что для выполнения цели `build` необходимо наличие файла `main.asm` в текущей директории. Это логично, так как без файла с исходным кодом создать программу не получится.

Также обратите внимание, что команды в правилах (в отличие от целей) начинаются с отступа, который обязательно должен создаваться символом табуляции (клавиша `Tab`), а не пробелами. В редакторе `mcedit` для создания табуляции нужно в меню (вызывается клавишей `F9`) выбрать `Настройка` `Общая` и убрать галочку в пункте «Симулировать неполную табуляцию» (или при редактировании Makefile использовать для создания отступа клавишу `Tab` дважды).

3.2.2. Основы работы с Midnight Commander

Midnight Commander (или просто `mc`) — это программа, которая позволяет просматривать структуру каталогов и выполнять основные операции по управлению файловой системой, т.е. `mc` является файловым менеджером.

Для активации оболочки Midnight Commander достаточно ввести в командной строке `mc` и нажать клавишу `Enter`.

Midnight Commander позволяет сделать работу с файлами более удобной и наглядной, хотя для работы с файлами и управления файловой системой могут быть использованы такие команды операционной системы, как `pwd`, `ls`, `cd`, `mv`, `mkdir`, `rmdir`, `cp`, `rm`, `cat`, `more` и т. д.

В Midnight Commander используются функциональные клавиши `F1` — `F10`, к которым привязаны часто выполняемые операции (табл. 3.1).

Таблица 3.1

Функциональные клавиши Midnight Commander

| Функциональная клавиша | Выполняемое действие |
|------------------------|---|
| F1 | вызов контекстно-зависимой подсказки |
| F2 | вызов меню, созданного пользователем |
| F3 | просмотр файла, на который указывает подсветка в активной панели |
| F4 | вызов встроенного редактора для файла, на который указывает подсветка в активной панели |
| F5 | копирование файла или группы отмеченных файлов из каталога, отображаемого в активной панели, в каталог, отображаемый на второй панели |
| F6 | перенос файла или группы отмеченных файлов из каталога, отображаемого в активной панели, в каталог, отображаемый на второй панели |
| F7 | создание подкаталога в каталоге, отображаемом в активной панели |
| F8 | удаление файла (подкаталога) или группы отмеченных файлов |
| F9 | вызов основного меню программы |
| F10 | выход из программы |

Команда меню **Дерево каталогов** отображает структуру каталогов файловой системы.

Меню **Команда** **Поиск файла** или комбинация клавиш **Alt** + **?** или **Esc** + **?** позволяют вызвать интерфейс для поиска файлов, текста в файлах и т.п. Поскольку вопросительный знак в английской раскладке находится на одной кнопке с символом **/**, то для вызова поиска следует нажимать комбинацию **Alt** + **Shift** + **/**.

Диалоговое окно «Поиск файла» позволяет найти на диске файл с заданным именем. После выбора этой команды запрашивается имя искомого файла и имя каталога, с которого необходимо начинать поиск.

Графическая кнопка **Дерево** позволяет выбрать начальный каталог поиска из дерева каталогов. Поле «Содержание» (Contents) позволяет задать регулярное выражение для поиска по правилам команды `egrep`. Нажатие

графической кнопки **Дальше** запускает процедуру поиска. Во время поиска его можно приостановить графической кнопкой **Остановить** и продолжить по кнопке **Продолжить**.

С помощью клавиш **↑** и **↓** можно просматривать список найденных файлов. Графическая кнопка **Перейти** используется для перехода в каталог, в котором находится подсвеченный файл, а кнопка **Повтор** служит для задания параметров нового поиска. Для выхода из режима поиска предназначена графическая кнопка **Выход**.

Следующие комбинации клавиш облегчают работу с Midnight Commander:

- **Ctrl + u** (или через меню **Команда** **Переставить панели**) меняет местами содержимое правой и левой панелей;
- **Ctrl + o** (или через меню **Команда** **Отключить панели**) скрывает или возвращает панели Midnight Commander, за которыми доступен для работы командный интерпретатор оболочки и выводимая туда информация.
- **Ctrl-x d** (или через меню **Команда** **Сравнить каталоги**) позволяет сравнить содержимое каталогов, отображаемых на левой и правой панелях.

Меню **Команда** **История команд** (аналог команды `history`) выводит список ранее выполнявшихся команд. Подсвеченную строку из истории можно скопировать в командную строку оболочки (перемещение подсветки — клавишами **↑** и **↓**, копирование — по клавише **Enter**).

Меню **Команда** **Редактировать файл расширений** позволяет редактировать файл `mc.ext`, в котором задаются правила связи определённых расширений файлов с инструментами их запуска или обработки (просмотра, редактирования или выполнения). Запуск выбранной программы осуществляется после выбора файла и нажатия клавиши **Enter**.

Меню **Команда** **Редактировать файл меню** предназначено для редактирования пользовательского меню (настройка клавиши **F2**).

Дополнительную информацию о Midnight Commander можно получить по команде `man mc`.

3.2.3. Структура программы на языке ассемблера NASM

Программа на языке ассемблера NASM, как правило, состоит из трёх секций: секция кода программы, секция иницированных (известных во время компиляции) данных и секция неинициализированных данных (тех, под которые во время компиляции только отводится память, а значение присваивается в ходе выполнения программы).

В NASM эти секции определяются с помощью директивы `SECTION`. Традиционно секция кода называется `.text`, секция иницированных данных — `.data`, а секция неинициализированных данных — `.bss`.

Таким образом, общая структура программы имеет следующий вид:

```

SECTION .data          ; Секция содержит переменные, для
...                   ; которых задано начальное значение

SECTION .bss           ; Секция содержит переменные, для
...                   ; которых не задано начальное значение

SECTION .text          ; Секция содержит код программы
GLOBAL _start

_start:               ; Точка входа в программу

...                   ; Текст программы

mov     eax,1          ; Системный вызов для выхода (sys_exit)
mov     ebx,0          ; Выход с кодом возврата 0 (без ошибок)
int     80h           ; Вызов ядра

```

Для объявления инициализированных данных в секции `.data` используются директивы `DB`, `DW`, `DD`, `DQ` и `DT`, которые резервируют память и указывают, какие значения должны храниться в этой памяти:

- `DB` (define byte) — определяет переменную размером в 1 байт;
- `DW` (define word) — определяет переменную размером в 2 байта (слово);
- `DD` (define double word) — определяет переменную размером в 4 байта (двойное слово);
- `DQ` (define quad word) — определяет переменную размером в 8 байт (учетверённое слово);
- `DT` (define ten bytes) — определяет переменную размером в 10 байт.

Директивы используются для объявления простых переменных и для объявления массивов. Для определения строк принято использовать директиву `DB` в связи с особенностями хранения данных в оперативной памяти.

Синтаксис директив определения данных следующий:

`<имя> DB <операнд> [, <операнд>] [, <операнд>]`

Примеры:

| | |
|-----------------------------|---|
| <code>a db 10011001b</code> | определяем переменную <code>a</code> размером 1 байт с начальным значением, заданным в двоичной системе счисления (на двоичную систему счисления указывает также буква <code>b</code> binary в конце числа) |
| <code>b db '!'</code> | определяем переменную <code>b</code> в 1 байт, инициализируемую символом <code>!</code> |
| <code>c db "Hello"</code> | определяем строку из 5 байт |
| <code>d dd -345d</code> | определяем переменную <code>d</code> размером 4 байта с начальным значением, заданным в десятичной системе счисления (на десятичную систему указывает буква <code>d</code> decimal в конце числа) |
| <code>h dd 0f1ah</code> | определяем переменную <code>h</code> размером 4 байта с начальным значением, заданным в шестнадцатеричной системе счисления (<code>h</code> — hexadecimal) |

Для объявления инициализированных данных в секции `.bss` используются директивы `resb`, `resw`, `resd` и другие, которые сообщают ассемблеру, что необходимо зарезервировать заданное количество ячеек памяти. Примеры их использования приведены в табл. 3.2.

Таблица 3.2

Директивы для объявления инициализированных данных

| Директива | Назначение директивы | Аргумент | Назначение аргумента |
|-------------------|---|-----------------------------|---|
| <code>resb</code> | Резервирование заданного числа однокбайтовых ячеек | <code>string resb 20</code> | По адресу с меткой <code>string</code> будет расположен массив из 20 однокбайтовых ячеек (хранение строки символов) |
| <code>resw</code> | Резервирование заданного числа двухбайтовых ячеек (слов) | <code>count resw 256</code> | По адресу с меткой <code>count</code> будет расположен массив из 256 двухбайтовых слов |
| <code>resd</code> | Резервирование заданного числа четырехбайтовых ячеек (двойных слов) | <code>x resd 1</code> | По адресу с меткой <code>x</code> будет расположено одно двойное слово (т.е. 4 байта для хранения большого числа) |

3.2.4. Элементы программирования

Описание инструкции `mov`. Инструкция языка ассемблера `mov` предназначена для дублирования данных источника в приёмнике. В общем виде эта инструкция записывается в виде

```
mov dst, src
```

Здесь `dst` — приёмник, `src` — источник.

Переслать значение из одной ячейки памяти в другую нельзя, для этого необходимо использовать две инструкции `mov`:

```
mov eax, x
```

```
mov y, eax
```

В табл. 3.3 приведены варианты использования `mov` с разными операндами:

Таблица 3.3

Варианты использования `mov` с разными операндами

| Пример | Пояснение |
|--|--|
| <code>mov eax, 403045h</code> <code>mov cx, [eax]</code> | пишет в <code>eax</code> значение 403045 помещает в регистр <code>cx</code> значение (размера <code>word</code>) из памяти, указанной в <code>eax</code> (403045) |
| <code>mov eax, ebx</code> <code>mov x, 0</code> <code>mov al, 1000h</code> | пересылает значение регистра <code>ebx</code> в регистр <code>eax</code> записывает в переменную <code>x</code> значение 0 ошибка — попытка записать 2-байтное число в 1-байтный регистр |
| <code>mov eax, cx</code> | ошибка — размеры операндов не совпадают |

Описание инструкции `int`. Инструкция языка ассемблера `int` предназначена для вызова прерывания с указанным номером. В общем виде она записывается в виде

`int n`

Здесь `n` — номер прерывания, принадлежащий диапазону 0–255. При программировании в Linux с использованием вызовов ядра (`sys_calls`) `n = 80h` (принято задавать в шестнадцатеричной системе счисления).

После вызова инструкции `int 80h` выполняется системный вызов какой-либо функции ядра Linux. При этом происходит передача управления ядру операционной системы. Чтобы узнать, какую именно системную функцию нужно выполнить, ядро извлекает номер системного вызова из регистра `eax`. Поэтому перед вызовом прерывания необходимо поместить в этот регистр нужный номер, например, выполнить `mov eax, 3` для системного вызова номер 3.

Многим системным функциям требуется передавать какие-либо параметры. По принятым в ОС Linux правилам эти параметры помещаются в порядке следования в остальные регистры процессора: `ebx`, `ecx`, `edx` и т. д. Если системная функция должна вернуть значение, то она помещает его в регистр `eax`.

Системные вызовы для обеспечения диалога с пользователем. Простейший диалог с пользователем требует наличия двух функций — вывода текста на экран и ввода текста с клавиатуры. Простейший способ вывести строку на экран — использовать системный вызов `write`, который аналогичен функции `write` из языка Си и предназначен для записи данных в файл. Этот системный вызов имеет номер 4, поэтому перед вызовом инструкции

int необходимо поместить значение 4 в регистр `eax`. Первым аргументом `write`, помещаемым в регистр `ebx`, задаётся дескриптор файла. Для вывода на экран в качестве дескриптора файла нужно указать 1 (это означает «стандартный вывод», т. е. вывод на экран).

Вторым аргументом задаётся адрес выводимой строки (помещаем его в регистр `ecx`, например, инструкцией `mov ecx, hello`). Строка может иметь любую длину. Последним аргументом (т. е. в регистре `edx`) должна задаваться максимальная длина выводимой строки (посмотрите, как это делалось в программе из предыдущей работы).

Для ввода строки с клавиатуры можно использовать аналогичный системный вызов `read`. Этот системный вызов имеет номер 3. Подробная информация о нём, предоставляемая командой `man 2 read`, показывает, что его аргументы — такие же, как у вызова `write`, только для «чтения» с клавиатуры используется файловый дескриптор 0 (стандартный ввод), а не 1 (стандартный вывод), как было при выводе на экран. Возвращаемое через регистр `eax` значение функции `read` — количество прочитанных с клавиатуры символов.

Например, следующая программа ждёт ввода строки и введённую строку сохраняет в область памяти, помеченную меткой `buf1`:

```

;-----
; Программа ввода строки с клавиатуры
;-----

SECTION .bss                ; Секция не инициированных данных
buf1:    RESB 80            ; Буфер размером 80 байт

SECTION .text               ; Код программы
GLOBAL _start              ; Начало программы

_start:

mov     eax, 3              ; Системный вызов для чтения (sys_read)
mov     ebx, 0              ; Дескриптор файла
                        ; 0 - стандартный ввод
mov     ecx, buf1           ; Адрес буфера под вводимую строку
mov     edx, 80             ; Длина вводимой строки

int     80h                ; Вызов ядра

mov     eax, 1              ; Системный вызов для выхода (sys_exit)
mov     ebx, 0              ; Выход с кодом возврата 0 (без ошибок)
int     80h                ; Вызов ядра

```


3.3. Порядок выполнения работы

1. Создайте в своём домашнем каталоге новый подкаталог с именем `lab03a` и файл `asdfg.asm`:

```
cd  
mkdir lab03a  
touch asdfg.asm
```
2. Пользуясь информацией, приведённой в теоретической части, напишите программу, работающую по следующему алгоритму:
 - (a) вывести приглашение типа «Введите строку:»;
 - (b) ввести строку с клавиатуры;
 - (c) вывести введённую строку на экран.
3. Получите исполняемый файл и проверьте его работу. На приглашение ввести строку введите свою фамилию.
4. Создайте в своём домашнем каталоге новый подкаталог `lab03b` и скопируйте в него созданный файл с текстом программы.
5. Скопируйте файл `asdfg.asm` в `lab03-1.asm`.
6. Оттранслируйте полученный текст программы в объектный файл по схеме `lab03-1.asm → o.o` и `asdfg.asm → w.o`.
7. Создайте для `make` файл с явными правилами получения исполняемых файлов двух написанных программ. Проверьте работу `make`.

3.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;
- 3) описание процесса выполнения задания. Для каждого действия, производимого в командной строке, в отчёт следует включить:
 - краткое описание действия;
 - вводимую команду или команды;
 - результаты выполнения команд и программ (снимок экрана);
- 4) листинги программ;
- 5) выводы, согласованные с целью работы.
- 6) ответы на контрольные вопросы.

3.5. Контрольные вопросы

1. Каково назначение утилиты `make`?
2. Где задаются правила поведения `make`?
3. Какое расширение у `Makefile`?
4. Какова структура программы на языке ассемблера `NASM`?
5. Для описания каких данных используются секции `bss` и `data` на языке ассемблера `NASM`?

6. Для чего используются компоненты `db`, `dw`, `dd`, `dq` и `dt` языка ассемблера NASM?
7. Какое произойдёт действие при выполнении инструкции `mov eax, esi`?
8. Для чего используется инструкция `int 80h`?

При ответах на вопросы используйте сведения из [1—3].

Список литературы

1. Расширенный ассемблер: NASM. — 2001. — URL: <https://www.opennet.ru/docs/RUS/nasm/>.
2. *Столяров А.* Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011.
3. *Куляс О. Л., Никитин К. А.* Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.

Лабораторная работа № 4. Работа с файлом листинга

4.1. Цель работы

Знакомство с назначением и структурой файла листинга. Изучение команд условного и безусловного переходов. Приобретение навыков написания программ с использованием переходов.

4.2. Предварительные сведения

4.2.1. Файл листинга и его назначение

Листинг (в рамках понятийного аппарата NASM) — это один из выходных файлов, создаваемых транслятором. Он имеет текстовый вид и нужен при отладке программы, так как кроме строк самой программы он содержит дополнительную информацию.

Обычно `nasm` создаёт в результате ассемблирования только объектный файл. Получить файл листинга можно, указав ключ `-l` и задав имя файла листинга в командной строке. Например:

```
nasm -l main.lst main.asm
```

Ниже приведён фрагмент файла листинга.

```
1  BITS32 ; Сообщаем компилятору, что код 32-битный
2  section .data
3  00000000 48656C6C6F20776F72- hello:      db 'Hello world!',10
4  00000009 6C64210A
5  helloLen: equ $-hello ; Длина строки
6
7  section .text
8  global _start
9  ; чтобы линкер смог найти метку и сделать её точкой входа в
   ↳ программу.
10
11 start:
12 00000000 B804000000 mov eax,4 ; Системный вызов для записи
   ↳ (sys_write)
13 00000005 BB01000000 mov ebx,1 ; Описатель файла 1 -
   ↳ стандартный вывод
14 0000000A B9[00000000] mov ecx,hello ; Адрес строки hello в ecx
15 0000000F BA0D000000 mov edx,helloLen ; helloLen - это
   ↳ константа
16 ; mov edx,[helloLen] для получения действительного значения
17 00000014 CD80      int 80h      ; Вызов ядра
18
```

```

19 00000016 B801000000 mov eax,1 ; Системный вызов для выхода
   ↳ (sys_exit)
20 0000001B BB00000000 mov ebx,0 ; Выход с кодом возврата 0
   ↳ (без ошибок)
21 00000020 CD80      int 80h      ; Вызовов ядра

```

4.2.2. Структура листинга

Строки в первой части листинга имеют следующую структуру (рис. 4.1).

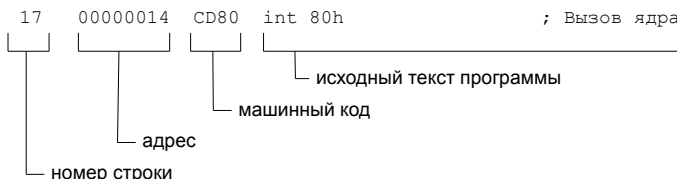


Рис. 4.1. Структура листинга

Все ошибки и предупреждения, обнаруженные при ассемблировании, транслятор выводит на экран, и файл листинга не создаётся.

Итак, структура листинга:

- *номер строки* — это номер строки файла листинга (нужно помнить, что номер строки в файле листинга может не соответствовать номеру строки в файле с исходным текстом программы);
- *адрес* — это смещение машинного кода от начала текущего сегмента;
- *машинный код* представляет собой ассемблированную исходную строку в виде шестнадцатеричной последовательности. (например, инструкция `int 80h` начинается по смещению `00000014` в сегменте кода; далее идёт машинный код, в который ассемблируется инструкция, то есть инструкция `int 80h` ассемблируется в `CD80` (в шестнадцатеричном представлении); `CD80` — это инструкция на машинном языке, вызывающая прерывание ядра);
- *исходный текст программы* — это просто строка исходной программы вместе с комментариями (некоторые строки на языке ассемблера, например, строки, содержащие только комментарии, не генерируют никакого машинного кода, и поля «смещение» и «исходный текст программы» в таких строках отсутствуют, однако номер строки им присваивается).

Допустимыми символами в метках являются буквы, цифры, а также следующие символы: `_`, `$`, `#`, `@`, `~`, `.`, `,` и `?`. Начинаться метка или идентификатор могут с буквы, `.`, `_` и `?`. Перед идентификаторами, которые пишутся как зарезервированные слова, нужно писать `$`, чтобы компилятор трактовал его верно (так называемое *экранирование*). Максимальная длина идентификатора 4095 символов.

4.2.3. Элементы программирования

Описание инструкции add. Схема команды целочисленного сложения `add` выполняет сложение двух операндов и записывает результат по адресу первого операнда. Команда `add` выглядит следующим образом:
`add` операнд_1, операнд_2

Описание инструкции sub. Схема команды целочисленного вычитания `sub` выглядит следующим образом:
`sub` операнд_1, операнд_2

- Работа команды включает два действия:
- выполнить вычитание: `операнд_1 = операнд_2 - операнд_1;`
 - установить флаги.
- Флаги, устанавливаемые командой, подробнее рассматриваются далее.

Описание инструкции cmp. Инструкция `cmp` является командой сравнения двух операндов и имеет такой же формат, как и команда вычитания:
`cmp` операнд_1, операнд_2

Команда `cmp`, так же как и команда вычитания, выполняет вычитание `операнд_2 - операнд_1`, но результат вычитания никуда не записывается и единственным результатом команды сравнения является формирование флагов, которые устанавливаются так же, как и при выполнении команды вычитания.

Описание команд условного перехода. Команды условного перехода, анализируя флаги из регистра флагов (рис. 4.2), вычисляют условие перехода.

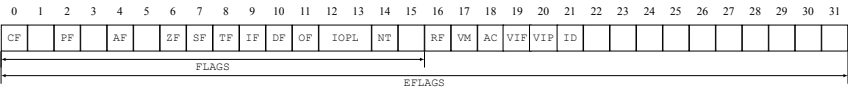


Рис. 4.2. Регистр флагов

Команда условного перехода имеет вид
`j<мнемоника перехода> label`
Мнемоника перехода связана со значением анализируемых флагов или со способом формирования этих флагов. Чаще всего для формирования флагов проверяется отношение двух операндов `операнд_1 <отношение> операнд_2` с помощью команды вычитания или команды сравнения.

Инструкции условной передачи управления представлены в табл. 4.1. Мнемоники, идентичные по своему действию, написаны в табл. 4.1 через дробь (например, `ja` и `jnb`). Программист выбирает, какую из них применить, чтобы получить более простой для понимания текст программы.

Таблица 4.1

Инструкции условной передачи управления

| Мнемокод | Флаги | Смысл |
|----------|-----------------------|-------------------------------|
| ja/jnbe | CF or ZF=0 | выше /не ниже и не равно |
| jae/jnb | CF=0 | выше или равно/не ниже |
| jb/jnae | CF=1 | ниже/не выше и не равно |
| jbe/jna | CF or ZF=1 | ниже или равно/не выше |
| je/jz | ZF=1 | равно/нуль |
| jne/jnz | ZF=0 | не равно/не нуль |
| jg/jnle | (SF xor OF) or ZF=0 | больше/не меньше и не равно |
| jge/jnl | SF xor OF=0 | больше или равно/не меньше |
| jl/jnge | (SF xor OF)=1 | меньше/не больше и не равно |
| jle/jng | ((SF xor OF) or ZF)=1 | меньше или равно/не больше |
| jp/jpe | PF=1 | есть паритет/паритет чётный |
| jnp/jpo | PF=0 | нет паритета/паритет нечётный |
| jc | CF=1 | перенос |
| jnc | CF=0 | нет переноса |
| jo | OF=1 | переполнение |
| jno | OF=0 | нет переполнения |
| jns | SF=0 | знак + |
| js | SF=1 | знак - |

Примечание: термины «выше» («a» от англ. «above») и «ниже» («b» от англ. «below») применимы для сравнения беззнаковых величин (адресов), а термины «больше» («g» от англ. «greater») и «меньше» («l» от англ. «lower») используются при учёте знака числа. Таким образом, мнемонику инструкции ja/jnbe можно расшифровать как «jump (переход) if above (если выше) / if not below equal (если не меньше или равно)».

4.3. Порядок выполнения работы

- Написать программу, работающую по следующему алгоритму:
 - вывести на экран запрос о времени дня, например «Полдень наступил?»;
 - принять с клавиатуры ответ (y/n);
 - если было введено n, выдать сообщение «Доброе утро», в противном случае — «Добрый день».
- Получить файл листинга и внимательно ознакомиться с его форматом и содержанием.

3. В любой инструкции с двумя операндами удалить один операнд и проасемблировать программу с получением файла листинга. Какие выходные файлы создаются в этом случае? Что добавляется в листинге?
4. Подробно объяснить содержимое трёх строк файла листинга по выбору.
5. Получить имя владельца и числовой код прав доступа для файлов, сгенерированных в результате выполнения работы. Расшифровать права доступа.
6. Изменить права доступа к исполняемому файлу, запретив его выполнение. Попытаться выполнить файл. Объяснить результат.
7. Разрешить выполнение исходного текста программы как исполняемого файла. Попытаться выполнить его и объяснить результат.

4.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;
- 3) описание результатов выполнения задания:
 - листинги программ (если они есть в задании);
 - краткое описание всех действий, сделанных при выполнении заданий;
 - результаты выполнения команд и программ (снимок экрана).
- 4) выводы, согласованные с целью работы;
- 5) ответы на контрольные вопросы.

4.5. Контрольные вопросы

1. Для чего нужен файл листинга NASM? В чём его отличие от текста программы?
2. Каков формат файла листинга NASM? Из каких частей он состоит? Каково назначение первой части?
3. Как в программах на ассемблере можно выполнить ветвление?

При ответах на вопросы используйте сведения из [1—3].

Список литературы

1. Расширенный ассемблер: NASM. — 2001. — URL: <https://www.opennet.ru/docs/RUS/nasm/>.
2. *Столяров А.* Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011.
3. *Куляс О. Л., Никитин К. А.* Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.

Лабораторная работа № 5. Отладчик GDB. Программирование цикла с переедресацией. Организация стека

5.1. Цель работы

Знакомство с методами отладки при помощи GDB [1] и его основными возможностями. Приобретение навыков написания программ с использованием циклов.

5.2. Предварительные сведения

5.2.1. Понятие об отладке

Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

- обнаружение ошибки;
- поиск её местонахождения;
- определение причины ошибки;
- исправление ошибки.

Этап обнаружения ошибки наступает при появлении неустранимого сбоя в программе. В этом случае операционная система завершает её работу.

Можно выделить следующие типы ошибок:

- синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка;
- семантические ошибки — являются логическими и приводят к тому, что программа запускается, отработывает, но не даёт желаемого результата;
- ошибки в процессе выполнения — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль).

Второй этап — поиск местонахождения ошибки. Некоторые ошибки обнаружить довольно трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга.

Третий этап — выяснение причины ошибки. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы.

Последний этап — исправление ошибки. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново.

5.2.2. Методы отладки

Наиболее часто применяют следующие методы отладки:

- создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые *диагностические сообщения*);
- использование специальных программ-отладчиков.

Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование *точек останова* и *выполнение программы по шагам*.

Пошаговое выполнение — это выполнение программы с остановкой после каждой строчки, чтобы программист мог проверить значения переменных и выполнить другие действия.

Точки останова — это специально отмеченные места в программе, в которых программа-отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды точек останова:

- Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмеченной программистом);
- Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его).

Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы.

5.2.3. Основные возможности отладчика GDB

GDB (GNU Debugger — отладчик проекта GNU) [1] работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки.

В этой работе мы познакомимся с самыми основными возможностями GDB.

Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя.

GDB может выполнять следующие действия:

- начать выполнение программы, задав всё, что может повлиять на её поведение;
- остановить программу при указанных условиях;
- исследовать, что случилось, когда программа остановилась;
- изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

5.2.3.1. Запуск отладчика GDB; выполнение программы; выход

Синтаксис команды для запуска отладчика имеет следующий вид:

```
gdb [опции] [имя_файла | ID процесса]
```

После запуска (gdb) выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (gdb) для ввода команд.

Далее приведён список некоторых команд GDB.

Справку о любой команде (gdb) можно получить, введя

```
(gdb) help [имя_команды]
```

Командой `file` можно указать имя исполняемого файла, который нужно отладить, если при запуске GDB оно не было указано:

```
(gdb) file <имя исполняемого файла>
```

Команда `run` (сокращённо `r`) — запускает отлаживаемую программу в оболочке GDB. Можно указать список аргументов программы, если в этом есть потребность. Также можно организовать перенаправление потоков ввода и вывода в другие файлы, например:

```
(gdb) run > outfile
```

Если точки останова не были установлены, то программа выполняется и выводятся сообщения:

```
(gdb) run
```

```
Starting program: test
```

```
Program exited normally.
```

```
(gdb)
```

Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др.

Команда `kill` (сокращённо `k`) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки:

```
Kill the program being debugged? (y or n) y
```

Если в ответ введено `y` (то есть «да»), отладка программы прекращается. Командой `run` её можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются.

Для выхода из отладчика используется команда `quit` (или сокращённо `q`):

```
(gdb) q
```

5.2.3.2. Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`.

Однако такая возможность есть не всегда, и в случае необходимости отладчик может дизассемблировать исполняемый код, изображая машинные команды в виде ассемблерных мнемоник.

Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду

```
(gdb) set disassembly-flavor intel
```

5.2.3.3. Точки останова

Информацию о командах этого раздела можно получить, введя `help breakpoints`

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка»:

```
(gdb) break *0x4000b5
```

В данном случае была установлена точка останова по адресу 4000b5, заданному в шестнадцатеричной системе счисления.

Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`). Команда `info` имеет много возможностей, о которых можно узнать с помощью команды `help`.

Пример:

```
(gdb) info breakpoints
Num      Type           Disp Enb Address                What
1        breakpoint    keep y  0x000000000004000b0  main.asm:10
(gdb) b 20
Breakpoint 2 at 0x4000b0: file main.asm, line 20.
(gdb) i b
Num      Type           Disp Enb Address                What
1        breakpoint    keep y  0x000000000004000b0  lab4.asm:10
2        breakpoint    keep y  0x000000000004000b0  lab4.asm:20
(gdb)
```

Для того чтобы сделать неактивной какую-нибудь ненужную точку останова, можно воспользоваться командой `disable`:

`disable breakpoint <номер точки останова>`

Обратно точка останова активируется командой `enable`:

`enable breakpoint <номер точки останова>`

Активна точка останова или нет можно узнать с помощью той же команды `info`. Если же точка останова в дальнейшем больше не нужна, она может быть удалена:

`(gdb) delete breakpoint <номер точки останова>`
или

`(gdb) d b <номер точки останова>`

Ввод этой команды без аргумента удалит *все* точки останова.

5.2.3.4. Пошаговая отладка

Информацию о командах этого раздела можно получить, введя

`(gdb) help running`

Для продолжения остановленной программы используется команда `continue (c)`:

`(gdb) c [аргумент]`

Выполнение программы будет происходить до следующей точки останова. В качестве аргумента может использоваться целое число N , которое указывает отладчику проигнорировать $N - 1$ точку останова (выполнение остановится на N -й точке).

Команда `step` (кратко `s`) позволяет выполнять программу по шагам, т.е. до тех пор, пока не будет достигнута следующая строка её кода:

`(gdb) s [аргумент]`

При указании в качестве аргумента целого числа N отладчик выполнит команду `step N` раз при условии, что не будет точек останова или выполнение программы не прервётся по другим причинам.

Команда `next` (кратко `n`) действует аналогично `step`, но вызов процедуры считается единой инструкцией:

`(gdb) n [аргумент]`

Аргумент N работает так же, как и для `step`.

Команда `finish` (или `fin`) выполняет программу до момента выхода из текущей процедуры (функции) и выводит значение, если эта функция его возвращает:

`(gdb) fin`

Команда `until` (или `u`) производит выполнение программы до тех пор, пока не будет достигнута строка с номером, большим текущего. Эту команду удобно применять при отладке циклов.

Команда `stepi` (или `si`) работает аналогично команде `step`, но выполняет не строку, а ровно одну машинную инструкцию:

`(gdb) si [аргумент]`

При использовании аргумента N будет выполнено N инструкций.

Команда `nexti` (или `ni`) аналогична `stepi`, но вызов процедуры трактуется отладчиком как одна инструкция, а не как передача управления на ещё один блок ассемблерного кода, который тоже должен быть пройден по шагам:

```
(gdb) ni [аргумент]
```

При использовании дизассемблированной программы, скомпилированной без информации для отладки, нет возможности выполнить «одну строку исходного кода» за отсутствием такового. В этом случае используются команды `stepi` и `nexti`.

5.2.3.5. Работа с данными программы в GDB

Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Для отображения содержимого памяти можно использовать команду `x`.

`x` адрес выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/nfu адрес`. Рассмотрим задание формата подробнее.

- `n` — счётчик повторений. Это десятичное целое число, по умолчанию 1. Он определяет, сколько ячеек памяти отобразить (считая в единицах `u`).
- `f` — формат отображения. Наиболее употребляемые варианты: `s` (строка, оканчивающаяся нулём), `i` (машинная инструкция), `x` (шестнадцатеричное число (значение по умолчанию)). Значение по умолчанию изменяется каждый раз, когда вы используете команду с указанным в ней форматом.
- `u` — размер отображаемых ячеек памяти. Наиболее распространены варианты `b` (байт), `h` (полуслово, 2 байта), `w` (машинное слово, 4 байта, принято по умолчанию), `g` (длинное слово, восемь байт).

Например, `x/4uh 0x63450` — это запрос на вывод четырёх полуслов (`h`) из памяти в формате беззнаковых десятичных целых (`u`), начиная с адреса `0x63450`.

С помощью команды `x` можно отображать также значения регистров (перед именем регистра обязательно ставится префикс `$`), например, `p/x $ax`.

Если существует необходимость часто выводить значение какого-либо выражения (например, чтобы следить за тем, как оно меняется), можно добавить его в *список автоматического отображения*. В таком случае GDB будет выводить его значение каждый раз при остановке программы. Для этого служит команда `display`, которой можно передавать такой же аргумент, как команде `x`. Команда `delete display номер` позволяет убрать элемент данных с заданным номером из списка отображения.

И наконец, изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си):

```
set {char}0x600155='s'
```

5.2.4. Элементы программирования

Индексный доступ к данным. Для индексного доступа применяют косвенную адресацию. В этом случае адрес ячейки памяти заносят в регистр и используют его в качестве указателя. Для этих целей можно применять любые регистры. Признаком косвенной адресации являются прямоугольные скобки.

Пример использования косвенной адресации:

```
; --- Регистровая адресация ---
    mov     eax, esi        ; Значение регистра esi
                             ; дублируется в eax

; --- Косвенная адресация ---
    mov     eax, [esi]      ; слово, расположенное
                             ; по адресу esi,
                             ; дублируется в регистре eax
```

Инструкции организации циклов. Для организации циклов существуют специальные инструкции. Для всех инструкций максимальное количество проходов задаётся в регистре `ecx`. Наиболее простой является инструкция `loop`. Она позволяет организовать безусловный цикл:

```
; --- Организация циклов ---
    mov     ecx, 100        ; Количество проходов
    mov     esi, Buf1       ; Смещение на первый буфер
    mov     ebx, Buf2       ; Смещение на второй буфер
NextStep:
    mov     al, BYTE [esi]   ; Перенос байта из одного
    mov     BYTE [ebx], al   ; буфера в другой
    inc     esi              ; Перемещение указателей к
    inc     ebx              ; следующим элементам буфера
    loop    NextStep         ; Повторить ecx раз
                             ; от метки NextStep
```

5.2.5. Организация стека

Стек — это структура данных, организованная по принципу LIFO («Last In — First Out» или «последним пришёл — первым ушёл»). Стек является частью архитектуры процессора и реализован на аппаратном уровне. Для работы со стеком в процессоре есть специальные регистры (`ss`, `bp`, `sp`) и команды.

Основной функцией стека является функция сохранения адресов возврата и передачи аргументов при вызове процедур. Кроме того, в нём выделяется память для локальных переменных и могут временно храниться значения регистров.

На рис. 5.1 показана схема организации стека в процессоре i8086.

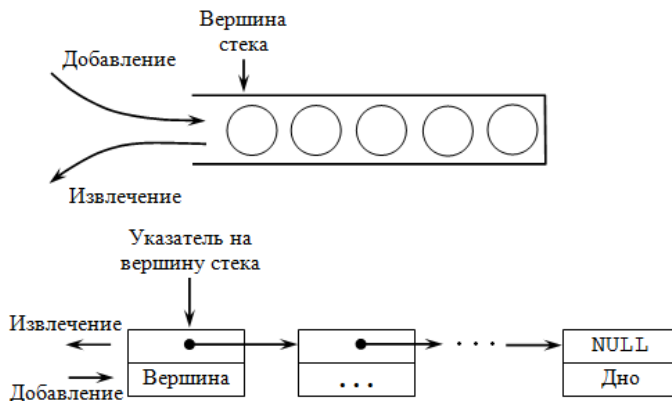


Рис. 5.1. Организация стека в процессоре i8086

Стек имеет вершину, адрес последнего добавленного элемента, который хранится в регистре `esp` (указатель стека). Противоположный конец стека называется дном. Значение, помещённое в стек последним, извлекается первым. При помещении значения в стек указатель стека уменьшается, а при извлечении — увеличивается.

Для стека существует две основные операции:

- добавление элемента в вершину стека (`push`);
- извлечение элемента из вершины стека (`pop`).

Добавление элемента в стек. Команда `push` размещает значение в стеке, т.е. помещает значение в ячейку памяти, на которую указывает регистр `esp`, после этого значение регистра `esp` увеличивается на 4. Данная команда имеет один операнд — значение, которое необходимо поместить в стек.

Примеры:

```
push -10      ; Поместить -10 в стек
push ebx      ; Поместить значение регистра ebx в стек
push [buf]    ; Поместить значение переменной buf в стек
push word [ax] ; Поместить в стек слово по адресу в ax
```

Существует ещё две команды для добавления значений в стек. Это команда `pusha`, которая помещает в стек содержимое всех регистров общего назначения в следующем порядке: `ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`. А также команда `pushf`, которая служит для перемещения в стек содержимого регистра флагов. Обе эти команды не имеют операндов.

Извлечение элемента из стека. Команда `pop` извлекает значение из стека, т.е. извлекает значение из ячейки памяти, на которую указывает регистр `esp`, после этого уменьшает значение регистра `esp` на 4. У этой команды также один операнд, который может быть регистром или переменной в памяти.

Нужно помнить, что извлечённый из стека элемент не стирается из памяти и остаётся как «мусор», который будет перезаписан при записи нового значения в стек.

Примеры:

```
pop eax      ; Поместить значение из стека в регистр eax
pop [buf]    ; Поместить значение из стека в buf
pop word [si] ; Поместить значение из стека
               ; в слово по адресу в si
```

Аналогично команде записи в стек существует команда `pora`, которая восстанавливает из стека все регистры общего назначения, и команда `porf` для перемещения значений из вершины стека в регистр флагов.

5.3. Порядок выполнения работы

1. Написать программу со следующим алгоритмом:
 - ввести с клавиатуры символьную строку в буфер;
 - изменить порядок следования символов в строке на противоположный; положение символа 10 (`\n`) остаётся без изменений;
 - вывести результат на экран;
 - завершить программу.
2. Загрузить программу в отладчик. Какими способами это можно сделать?
3. Просмотреть дизассемблированный код программы с помощью команды `disassemble _start`.
4. Переключить дизассемблер GDB с синтаксиса АТТ на синтаксис Intel и снова выполнить показ дизассемблированного кода. Найти три отличия. Переписать адрес второй инструкции в формате `0x12345678`.
5. Установить точку останова на второй инструкции, указав её.
6. Выполнить программу. Что произошло?
7. Выполнить программу по шагам.
8. Посмотреть содержимое регистров в окне с помощью команды `info r`.
9. Выполнить программу до места заполнения входного буфера. Вывести содержимое входного буфера в шестнадцатеричном формате и в символьном виде (команда `x`).
10. Выполнить 2 прохода цикла по шагам, контролируя значения регистров. Какие регистры изменяются в цикле?
11. Изменить содержимое выходного буфера с помощью команды `set`. Вводить данные как символы и как десятичные числа.

5.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;
- 3) описание результатов выполнения задания:
 - листинги программ (если они есть в задании);
 - краткое описание всех действий, сделанных при выполнении заданий;
 - результаты выполнения команд и программ (снимок экрана).
- 4) выводы, согласованные с целью работы;
- 5) ответы на контрольные вопросы.

5.5. Контрольные вопросы

1. Для чего нужен отладчик?
2. Расшифруйте и объясните следующие термины: breakpoint, watchpoint, checkpoint, catchpoint и call stack.
3. Объясните назначение отладочной информации и как нужно компилировать программу, чтобы в ней присутствовала отладочная информация.
4. Напишите 5 команд отладчика GDB (запуск, поставить точку останова с условием, продолжить, распечатать локальные переменные, завершить работу отладчика).
5. В чём заключается различие прямых и косвенных режимов адресации?
6. Как организовать цикл с помощью команд условных переходов, не прибегая к специальным командам управления циклами?
7. Дайте определение понятия «стек».
8. Как осуществляется порядок выборки содержащихся в стеке данных?

Список литературы

1. GDB: The GNU Project Debugger. — URL: <https://www.gnu.org/software/gdb/>.

Лабораторная работа № 6. Отладчик EDB. Понятие подпрограммы

6.1. Цель работы

Знакомство с методами отладки при помощи EDB и его основными возможностями. Приобретение навыков написания программ с использованием подпрограмм.

6.2. Предварительные сведения

6.2.1. Основные возможности отладчика EDB

Отладчик Evan's Debugger (EDB), как и любой другой отладчик, позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент краха, однако реализует эти возможности через наглядный графический интерфейс. Синтаксис команды для запуска отладчика имеет следующий вид:

```
edb [ --attach <ID процесса> ] [ --run <имя_файла> (аргументы) ]
```

После запуска появляется графическое окно EDB, разделённое на четыре основные части: дизассемблер, стек, дамп памяти с вкладками, содержимое регистров (рис. 6.1).

Всю информацию и данные Evan's Debugger отображает в меню и в окнах. Используются различные виды окон в зависимости от того, какого типа информация в них отображается. Все окна открываются и закрываются с помощью команд меню (или активных клавиш, соответствующих этим командам).

Окно регистров (Registers) показывает состояние регистров и флагов процессора, а также позволяет изменять их значения с помощью двойного нажатия мышкой. С помощью команд всплывающего (локального) меню можно перейти по адресу, хранящемуся в выбранном регистре, в окне стека или окне дампа памяти.

Окно дампа памяти (Data Dump) показывает построчное содержимое области памяти, которое представлено в виде шестнадцатеричных байтов, слов и двойных слов. Информацию, отображённую в данном окне, можно использовать для просмотра исходных данных. Во всплывающем меню имеются команды, которые позволяют модифицировать отображаемые данные, менять формат их отображения на экране и манипулировать блоками данных.

Окно стека (Stack) показывает текущее состояние стека, причём область первой вызванной функции будет находиться на дне стека, а всех

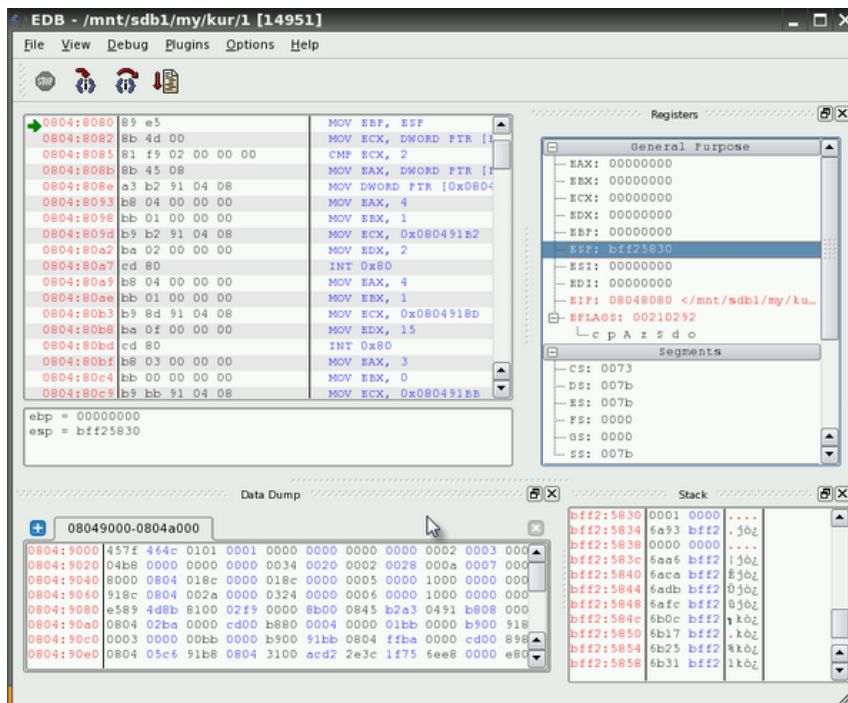


Рис. 6.1. Графическое окно EDB

последующих вызванных функций — в направлении вершины стека в последовательности их вызова. Во всплывающем меню имеются команды, которые позволяют модифицировать отображаемые данные, менять формат их отображения на экране, переходить по указанному адресу, по адресу из регистров `ebp` или `esp`.

6.2.1.1. Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`.

Однако такая возможность есть не всегда, и в случае необходимости отладчик может дизассемблировать исполняемый код, изображая машинные команды в виде ассемблерных мнемоник.

Напомним, что существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT. По умолчанию в дизассемблере EDB принят режим Intel.

6.2.1.2. Точки останова

Установить точку останова можно путём двойного клика мышкой на нужной инструкции. Если точка останова установилась, напротив инструкции появится красная отметка (рис. 6.2).



Рис. 6.2. Установка точки останова

Информацию о всех установленных точках останова можно посмотреть с помощью Breakpoint Manager, нажав **Ctrl** + **M** или открыв меню **Plugins** **BreakpointManager** **Breakpoints** (рис. 6.3).

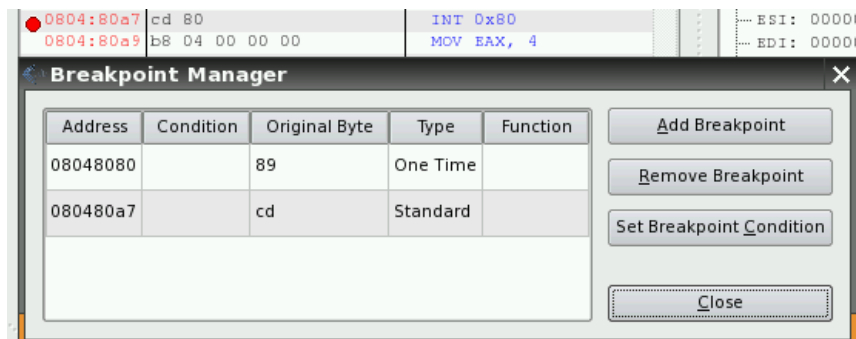


Рис. 6.3. Информация о всех установленных точках останова

6.2.1.3. Возобновление выполнения, пошаговая отладка

Команда **Run** (клавиша **F9**) продолжает выполнение остановленной программы. Выполнение будет происходить, пока не встретится точка останова или программа не будет выполнена полностью.

Команда **Step Into** (клавиша **F7**) приводит к выполнению программы до тех пор, пока не будет достигнута следующая строка её кода. Вызов

процедуры трактуется отладчиком не как одна инструкция, а как передача управления на ещё один блок ассемблерного кода, который тоже должен быть пройден по шагам.

Команда *Step Over* (клавиша **F8**) приводит к выполнению программы до тех пор, пока не будет достигнута следующая строка её кода. В отличие от *Step Into* вызов процедуры считается единой инструкцией.

6.2.2. Элементы программирования

6.2.2.1. Понятие подпрограммы

Подпрограмма — это, как правило, функционально законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом.

Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

6.2.2.2. Инструкция *call* и инструкция *ret*

Основные моменты выполнения подпрограммы иллюстрируются на рис. 6.4.

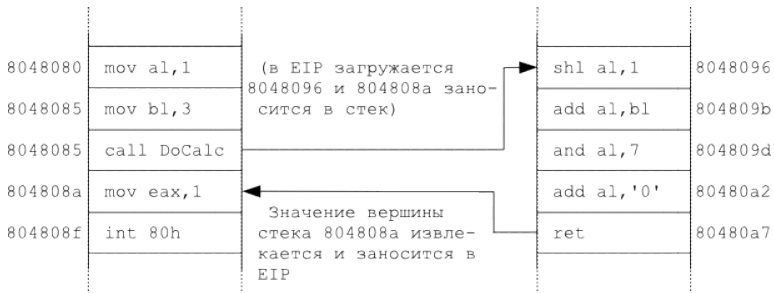


Рис. 6.4. Основные моменты выполнения подпрограммы

Для вызова подпрограммы из основной программы используется инструкция *call*, которая заносит адрес следующей инструкции в стек и загружает в регистр *eip* адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы.

Подпрограмма завершается инструкцией *ret*, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией *call*, и заносит


```

_start:
    mov     ecx, ask1
    mov     edx, ask1_len
    call    Print_string    ; Вызов подпрограммы Print_string

    mov     ecx, buf1
    mov     edx, 6
    call    Enter_string    ; Вызов подпрограммы Enter_string

    mov     ecx, result
    mov     edx, result_len
    call    Print_string    ; Вызов подпрограммы Print_string

    mov     ecx, buf1
    mov     edx, 6
    call    Print_string    ; Вызов подпрограммы Print_string

    mov     eax, 1          ; Системный вызов для выхода
                           ; (sys_exit)
    mov     ebx, 0          ; Выход с кодом возврата 0
                           ; (без ошибок)
    int     80h            ; Вызов ядра

```

Подпрограмма PrintString может печатать любую строку, на которую укажет регистр `ecx`.

6.2.2.3. Команды деления

Инструкция `div` выполняет беззнаковое деление и имеет следующий синтаксис:

```
div <делитель>
```

В качестве делителя может выступать область памяти или регистр общего назначения. Эта команда не работает непосредственно с числами.

```
div 3    ; неверно
```

Алгоритм работы команды `div` зависит от размера делителя.

Беззнаковое 16-битового операнда на 8-битовый операнд. Если делитель — это байт (8-бит), то делимое должно быть записано в регистре `ax`, результат деления будет записан в регистр `al`, а 8-битовый остаток — в `ah`, т.е. $al = ax / \text{делитель}$.

Например, после выполнения инструкций

```

mov     ax, 31
mov     dl, 15
div     dl

```

результат 2 (31/15) будет записан в регистр `al`, а остаток 1 (остаток от деления 31/15) — в регистр `ah`.

Так как частное представляет собой 8-битовое значение, то результат деления 16-битового операнда на 8-битовый операнд не должен превышать 255. Если частное больше, то генерируется ошибка деления на 0. Например, выполнение следующих инструкций

```
mov ax,0ffffh
mov bl,1
div bl
```

генерирует ошибку деления на 0.

Беззнаковое деление 32-битового операнда на 16-битовый операнд. Если делитель — это слово (16-бит), то делимое должно записываться в регистрах `dx:ax`, а результат деления будет равен $ax = (dx\ ax) / \text{делитель}$.

Например, в результате выполнения инструкций

```
mov ax,2 ; загрузить в регистровую
mov dx,1 ; пару \verb!dx:ax! 10002h
mov bx,10h
div bx
```

в регистр `ax` запишется частное 1000h (результат деления 10002h на 10h), а в регистр `dx` — 2 (остаток от деления).

При делении имеет значение, являются ли операнды знаковыми или беззнаковыми. Для деления операндов без знака используется инструкция `div`, а для деления знаковых операндов — `idiv`.

6.2.2.4. Способ перевода числа в десятичную символьную запись

Ввод информации с клавиатуры и вывод её на экран осуществляется в символьном виде. Кодирование этой информации производится согласно кодовой таблице символов, где каждый символ (в простейшем случае) кодируется одним байтом. Однако в памяти компьютера любые числа, над которыми можно производить математические операции, записаны в двоичной системе счисления, и для вывода на экран необходимо преобразовать двоичное число в его символьную запись (а при вводе с клавиатуры — выполнить обратное преобразование).

Любое число X в позиционной системе счисления представляется в виде суммы произведений:

$$X = a_n p_n + a_{n-1} p_{n-1} + \dots + a_0 + p_0,$$

здесь X — это число в системе с основанием p , имеющее $n + 1$ цифру в целой части.

Так, при переводе кода введённого символа в десятичную систему (например, чтобы вывести на экран не символ, а численное значение его кода) надо разложить число на слагаемые, содержащие степени числа 10. Перевод кода символа производится путём последовательного деления на основание 10 с выделением остатков от деления до тех пор, пока частное не станет

меньше делителя. Выписывая остатки от деления справа налево, получаем десятичную запись числа.

Далее приведена подпрограмма преобразования десятичного числа в строку.

```
convert:
    xor ecx,ecx    ; ecx = 0
    xor ebx,ebx    ; ebx = 0
    mov bl,10      ; ebx = 010, основание системы счисления

.divide:
    xor edx,edx    ; edx = 0
    div ebx        ; Делим eax на ebx, частное в eax,
                  ; остаток в edx
    add dl,'0'     ; Добавляем ASCII-код цифры 0 к остатку
    push edx       ; Сохраняем в стеке
    inc ecx        ; Увеличиваем счетчик цифр в стеке
    cmp eax,0      ; Закончили? (частное равно 0?)
    jnz .divide    ; Если не 0, переходим к .divide.
                  ; Иначе число уже преобразовано,
                  ; цифры сохранены в стеке,
                  ; ecx содержит их количество

.reverse:
    pop eax        ; Вытаскиваем цифру из стека
    stosb          ; Записываем al по адресу, содержащемуся
                  ; в edi, увеличиваем edi на 1
    loop .reverse  ; ecx=ecx-1, переходим, если ecx не равно 0
ret               ; Да? Возвращаемся
```

6.3. Порядок выполнения работы

1. Написать программу со следующим алгоритмом:

- ввести символ с клавиатуры;
- преобразовать полученный код в десятичную символьную запись;
- вывести символ и его код.

Перевод числа в десятичную символьную запись оформить в виде подпрограммы.

2. Загрузить программу в отладчик. Это можно сделать двумя способами: написать в командной строке `edb --run имя_программы` или запустить `edb` и выбрать программу через меню **Open** **File**.
3. Выполнить программу по шагам, нажимая кнопку **Step Over** панели инструментов или клавишу **F7** (находясь в основном окне отладчика), до конца.
4. Поместить в программу точку останова на инструкции, следующей после ввода символа с клавиатуры, — щёлкнув правой кнопкой по нужной строке дизассемблированного кода и выбрав пункт **Add Breakpoint** всплывающего меню. Выполнить программу до точки останова, нажав клавишу **F9** или кнопку **Run** панели инструментов. Имейте в виду, что ввод текста

с клавиатуры в выполняемую программу *осуществляется в отдельном окне EDB Output*, а не в основном окне отладчика.

5. Вывести в окне дампа памяти содержимое входного буфера, щёлкнув в окне Data Dump правой кнопкой мыши и выбрав пункт **Goto Address** всплывающего меню. Адрес вводить в шестнадцатеричной нотации Си (начиная с символов 0x).
6. Зайти в процедуру перевода числа в десятичную запись. Выполнить 2 прохода цикла по нажатию клавиши **F7** (Step Into), контролируя значения регистров. Какие регистры изменяются в цикле?
7. Остальные проходы цикла выполнить по нажатию клавиши **F8** (Step Over). В чём разница?
8. Определить физический адрес выходного буфера в ОЗУ.
9. Вывести ячейки памяти, соответствующие выходному буферу, в окне Data Dump в шестнадцатеричном и в символьном виде.

6.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;
- 3) описание результатов выполнения задания:
 - листинги программ (если они есть в задании);
 - краткое описание всех действий, сделанных при выполнении заданий;
 - результаты выполнения команд и программ (снимок экрана);
- 4) выводы, согласованные с целью работы;
- 5) ответы на контрольные вопросы.

6.5. Контрольные вопросы

1. Какие языковые средства используются в ассемблере для оформления и активизации подпрограмм?
2. Опишите структуру подпрограммы на языке ассемблера.
3. Объясните механизм вызова подпрограмм.
4. Как используется стек для обеспечения взаимодействия между вызывающей и вызываемой процедурами?
5. Каково назначение операнда в команде `ret`?

При ответах на вопросы рекомендуется воспользоваться информацией из источников [1—4].

Список литературы

1. Расширенный ассемблер: NASM. — 2001. — URL: <https://www.opennet.ru/docs/RUS/nasm/>.

2. *Столяров А.* Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011.
3. The NASM documentation. — 2017. — URL: <https://www.nasm.us/docs.php>.
4. *Куляс О. Л., Никитин К. А.* Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.

Лабораторная работа № 7. Побитовые операции

7.1. Цель работы

Приобретение навыков написания программ с использованием логических операций и операций побитового сдвига.

7.2. Предварительные сведения

7.2.1. Команды сдвига

Команды сдвига позволяют выполнять действия над отдельными битами операндов. Все команды сдвига перемещают биты в поле операндов влево или вправо. При выполнении команд сдвига флаг CF всегда содержит значение последнего выдвинутого бита. Рассмотрим следующие команды сдвига:

`shr операнд, счётчик_сдвигов ; логический (беззнаковый) сдвиг вправо`

`shl операнд, счётчик_сдвигов ; логический (беззнаковый) сдвиг влево`

В следующем примере команда `shr` сдвигает содержимое регистра `al` вправо на 1 бит. Выдвинутый в результате один бит попадает в флаг CF, а самый левый бит регистра `al` заполняется нулём.

```
mov al,10110111b ; в al содержится 10110111
shr al,1          ; в al содержится 01011011, CF=1
```

При сдвигах влево правые биты заполняются нулями.

Сдвиг влево на один двоичный разряд часто используется для удваивания чисел, а сдвиг на один разряд вправо — для деления на 2. Операции сдвига выполняются намного быстрее, чем команды умножения или деления.

Количество позиций, на которые должен быть выполнен сдвиг, задаётся вторым аргументом инструкции сдвига, и может быть либо задано константой, либо регистром `cl`: например, `shl ax, 12` или `shl eax, cl`.

7.2.2. Команды циклического сдвига

Циклический сдвиг — это операция сдвига, при которой бит, выдвинутый с одного «конца» числа, занимает освободившийся разряд с другой стороны этого же числа. Далее перечислены команды циклического сдвига:

`ror ; Циклический сдвиг вправо`

`rol ; Циклический сдвиг влево`

`rcr ; Циклический сдвиг вправо с переносом`

`rcl ; Циклический сдвиг влево с переносом`

В командах `rcr` и `rc1` в сдвиге участвует флаг `CF`. Выдвигаемый из регистра бит заносится в флаг `CF`, а прежнее значение `CF` при этом поступает в освободившуюся позицию в регистре.

7.2.3. Команды работы с битами операндов

Команды `and`, `or`, `xor` и `test` являются командами логических операций. Эти команды имеют два операнда и используются для сброса, установки и проверки бит. Размерность операндов должна быть одинаковой. Например, если размерность операндов равна слову (16 бит), то логическая операция выполняется сначала над нулевыми битами операндов, и далее над всеми битами с первого по пятнадцатый. Результат записывается на место первого операнда. Исключение составляет команда `test`. Команда `test` действует аналогично команде `and`, но результат не записывает на место первого операнда, а устанавливает только флаги во флаговом регистре. Это даёт возможность анализировать отдельные биты, не изменяя операнд.

| Таблица истинности для | <code>and</code> | <code>or</code> | <code>xor</code> |
|------------------------|------------------|-----------------|------------------|
| Значение операнда 1 | 0101 | 0101 | 0101 |
| Значение операнда 2 | 0011 | 0011 | 0011 |
| Результат операции | 0001 | 0111 | 0110 |

Команда `or` даёт в результате 1, если хотя бы один из сравниваемых битов равен 1, в случае если сравниваемые биты равны 0, то результат — 0. Команда `and` даёт в результате 1, если оба из сравниваемых битов равны 1, во всех остальных случаях результат — 0. Команда `xor` даёт результат 1, если один из сравниваемых битов равен 0, а другой 1, если сравниваемые биты одинаковы, то результат — 0.

Будем рассматривать байтную переменную `flags` как восемь независимых флагов, которые необходимо сбрасывать, устанавливать и анализировать в какой-либо программе. Для этого в правый операнд заносим «маску» в двоичной системе счисления.

Рассмотрим пример установки второго бита с использованием команды `or`:

```
or flags, 00000100b ; второй бит = 1, остальные
                     ; без изменений
```

Сбросить второй бит можно командой `and`, но маска должна быть инверсной:

```
and flags, 11111011b ; второй бит = 0, остальные
                     ; без изменений
```

Для того чтобы проверить, установлен ли нужный бит, применяется команда `test`:

```
test flags, 100b ; переменная flags не изменилась
jz ... ; изменены zf, sf, pf
```

Для сброса всех бит можно использовать команду `xor`:

```
xor ax, ax ; ax --- обнулён
```

Команда `not` устанавливает инвертированное значение бит в байте, в слове, в регистре или в памяти, т.е. единицы становятся нулями, а нули — единицами. Если, например, регистр `b1` содержит `1010 0101`, то команда `not b1` изменяет это значение на `0101 1010`.

7.3. Порядок выполнения работы

Написать программу со следующим алгоритмом:

- вывести приглашение;
- ввести с клавиатуры строку (предполагается, что она содержит десятичные цифры и любые буквы);
- найти во введённой строке все цифры и для каждой найденной цифры установить в «1» в регистре `ax` бит, номер которого равен этой цифре;
- вывести на экран содержимое регистра `ax` в виде нулей и единиц;
- объяснить полученный результат.

Ниже приведён пример подпрограммы, устанавливающий в 1 биты в `buf2`, соответствующие значениям символов массива `buf1`.

```
bit:
    mov ecx, 0
    mov esi, buf1 ; в esi записываем адрес входного буфера

lp1:
    inc esi
    mov al, [esi] ; в al первый символ из buf1
    cmp al, 10 ; конец строки?
    je end
    sub al, '0' ; проверка
    cmp al, 0 ; является ли
    JB lp1 ; символ цифрой,
    cmp al, 9 ; иначе переход
    JA lp1 ; на метку end
    mov cl, al
    mov edx, 1
    shl edx, cl ; сдвиг единичного бита на значение
    or [buf2], edx ; регистра cl, и запись в buf2
    jmp lp1 ; переход к следующему символу

end:
ret
```

7.4. Содержание отчёта

Отчёт должен включать:

- 1) титульный лист;
- 2) формулировку цели работы;
- 3) описание результатов выполнения задания:
 - листинги программ (если они есть в задании);

- краткое описание всех действий, сделанных при выполнении заданий;
 - результаты выполнения команд и программ (снимок экрана).
- 4) выводы, согласованные с целью работы;
 - 5) ответы на контрольные вопросы.

7.5. Контрольные вопросы

1. Перечислите команды сдвига с их краткой характеристикой.
2. В чём отличие команд логического и циклического сдвига?
3. Можно ли умножение и деление заменить командами сдвига?
4. Перечислите логические команды с их краткой характеристикой.
5. Какой командой можно заменить команду `mov AX, 0`?

При ответах на вопросы рекомендуется воспользоваться информацией из источников [1—4].

Список литературы

1. Расширенный ассемблер: NASM. — 2001. — URL: <https://www.opennet.ru/docs/RUS/nasm/>.
2. *Столяров А.* Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011.
3. The NASM documentation. — 2017. — URL: <https://www.nasm.us/docs.php>.
4. *Куляс О. Л., Никитин К. А.* Курс программирования на ASSEMBLER. — М. : Солон-Пресс, 2017.

Учебно-методический комплекс

Рекомендуется для направлений подготовки

01.03.02 — «Прикладная математика и информатика»

02.03.01 — «Математика и компьютерные науки»

02.03.02 — Фундаментальная информатика и информационные технологии

09.03.03 — «Прикладная информатика»

38.03.05 — «Бизнес-информатика»

Квалификация (степень) выпускника: бакалавр

Программа дисциплины

1. Цели и задачи дисциплины

Дисциплина «Архитектура вычислительных систем» для направлений подготовки 38.03.05 «Бизнес-информатика», 02.03.02 «Фундаментальная информатика и информационные технологии», 09.03.03 «Прикладная информатика» и дисциплина «Архитектура компьютеров» для направлений подготовки 01.03.02 «Прикладная математика и информатика», 02.03.01 «Математика и компьютерные науки» имеют целью введение учащихся в предметную область архитектуры компьютеров и вычислительных систем, изложение основных теоретических концепций, положенных в основу построения современных вычислительных систем.

В процессе преподавания дисциплины решаются следующие задачи:

- изучение принципов построения и архитектур современных компьютерных систем;
- изучение аппаратной части компьютера, его технических характеристик и функциональных возможностей;
- изучение основ программирования на низкоуровневом языке Assembler.

2. Место дисциплины в структуре ОП ВО

В табл. П.1 приведены предшествующие и последующие дисциплины, направленные на формирование компетенций обучающегося в соответствии с матрицей компетенций ОП ВО по направлению 02.03.02.

Таблица П.1

Предшествующие и последующие дисциплины, направленные на формирование компетенций по направлению 02.03.02

| № п/п | Шифр компетенции | Предшествующие дисциплины | Последующие дисциплины (группы дисциплин) |
|----------------------------------|------------------|---------------------------|---|
| Общепрофессиональные компетенции | | | |
| 1. | ОПК-1 | — | Операционные системы |

Описание компетенций для направления 02.03.02:

ОПК-1 — способность использовать базовые знания естественных наук, математики и информатики, основные факты, концепции, принципы теорий, связанных с фундаментальной информатикой и ИТ.

В табл. П.2 приведены предшествующие и последующие дисциплины, направленные на формирование компетенций обучающегося в соответствии с матрицей компетенций ОП ВО по направлению 09.03.03.

Таблица П.2

**Предшествующие и последующие дисциплины, направленные
на формирование компетенций по направлению 09.03.03**

| № п/п | Шифр ком- петен- ции | Предшествующие дисциплины | Последующие дисциплины (группы дисциплин) |
|----------------------------------|-------------------------------|------------------------------|--|
| Общепрофессиональные компетенции | | | |
| 1. | ОПК-4 | — | Операционные системы |

Описание компетенций для направления 09.03.03:

ОПК-4 — способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учётом основных требований информационной безопасности.

В табл. П.3 приведены предшествующие и последующие дисциплины, направленные на формирование компетенций обучающегося в соответствии с матрицей компетенций ОП ВО по направлению 38.03.05.

Таблица П.3

**Предшествующие и последующие дисциплины, направленные
на формирование компетенций по направлению 38.03.05**

| № п/п | Шифр ком- петен- ции | Предшествующие дисциплины | Последующие дисциплины (группы дисциплин) |
|---|-------------------------------|------------------------------|--|
| Общепрофессиональные компетенции | | | |
| 1. | ОПК-3 | — | Операционные системы |
| Профессиональные компетенции — производственно-технологическая деятельность | | | |
| 1. | ПК-30 | — | Операционные системы |

Описание компетенций для направления 38.03.05:

ОПК-3 — способность работать с компьютером как средством управления информацией, работать с информацией из различных источников, в том числе в глобальных компьютерных сетях;

ПК-30 — владеть базовыми знаниями и навыками использования в профессиональной деятельности операционных систем, платформенных окружений, сетевых технологий.

В табл. П.4 приведены предшествующие и последующие дисциплины, направленные на формирование компетенций обучающегося в соответствии с матрицей компетенций ОП ВО по направлению 01.03.02.

Таблица П.4

Предшествующие и последующие дисциплины, направленные на формирование компетенций по направлению 01.03.02

| № п/п | Шифр компетенции | Предшествующие дисциплины | Последующие дисциплины (группы дисциплин) |
|----------------------------------|------------------|---------------------------|---|
| Общепрофессиональные компетенции | | | |
| 1. | ОПК-1 | — | Операционные системы |

Описание компетенций для направления 01.03.02:

ОПК-1 — способность использовать базовые знания естественных наук, математики и информатики, основные факты, концепции, принципы теорий, связанных с прикладной математикой и информатикой.

В табл. П.5 приведены предшествующие и последующие дисциплины, направленные на формирование компетенций обучающегося в соответствии с матрицей компетенций ОП ВО по направлению 02.03.01.

Таблица П.5

Предшествующие и последующие дисциплины, направленные на формирование компетенций по направлению 02.03.01

| № п/п | Шифр компетенции | Предшествующие дисциплины | Последующие дисциплины (группы дисциплин) |
|----------------------------------|------------------|---------------------------|---|
| Общепрофессиональные компетенции | | | |
| 1. | ОПК-2 | — | Операционные системы |

Описание компетенций для направления 02.03.01:

ОПК-4 — способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учётом основных требований информационной безопасности.

3. Требования к результатам освоения дисциплины

Процесс изучения дисциплины «Архитектура вычислительных систем» направлен на формирование следующих компетенций:

- для направления подготовки 38.03.05 «Бизнес-информатика» — ОПК-3, ПК-30;
- для направления подготовки 02.03.02 «Фундаментальная информатика и информационные технологии» — ОПК-1;
- для направления подготовки 09.03.03 «Прикладная информатика» — ОПК-4.

Процесс изучения дисциплины «Архитектура компьютеров» направлен на формирование следующих компетенций:

- для направления подготовки 02.03.01 «Математика и компьютерные науки» — ОПК-2;
- для направления подготовки 01.03.02 «Прикладная математика и информатика» — ОПК-1.

В результате изучения дисциплины студент должен:

Знать:

- основные характеристики и области применения компьютеров и вычислительных систем;
- функциональную и структурную организацию основных устройств компьютера.

Уметь:

- использовать полученные знания при практической работе на персональном компьютере;
- эффективно использовать системные ресурсы компьютера;
- разрабатывать программы, с использованием языков программирования низкого уровня.

Владеть:

- способностью использовать современные средства компьютеров и вычислительных систем;
- навыками программирования на языке ассемблера.

4. Объем дисциплины и виды учебной работы

Для направлений подготовки 38.03.05 «Бизнес-информатика», 09.03.03 «Прикладная информатика» общая трудоёмкость дисциплины составляет 3 зачётные единицы.

| Вид учебной работы | Всего часов | Семестры |
|---------------------------------------|-------------|------------|
| | | 1 |
| Аудиторные занятия (всего) | 34 | 34 |
| В том числе: | | |
| <i>Лекции</i> | 17 | 17 |
| <i>Практические занятия (ПЗ)</i> | - | - |
| <i>Семинары (С)</i> | - | - |
| <i>Лабораторные работы (ЛР)</i> | 17 | 17 |
| Самостоятельная работа (всего) | 74 | 74 |
| Общая трудоёмкость: | | |
| час. | 108 | 108 |
| зач. ед. | 3 | 3 |

Для направлений подготовки 01.03.02 «Прикладная математика и информатика», 02.03.01 «Математика и компьютерные науки», 02.03.02 «Фундаментальная информатика и информационные технологии» общая трудоёмкость дисциплины составляет 3 зачётные единицы.

| Вид учебной работы | Всего часов | Семестры |
|---------------------------------------|-------------|------------|
| | | 1 |
| Аудиторные занятия (всего) | 51 | 51 |
| В том числе: | | |
| <i>Лекции</i> | 17 | 17 |
| <i>Практические занятия (ПЗ)</i> | - | - |
| <i>Семинары (С)</i> | - | - |
| <i>Лабораторные работы (ЛР)</i> | 34 | 34 |
| Самостоятельная работа (всего) | 57 | 57 |
| Общая трудоёмкость: | | |
| час. | 108 | 108 |
| зач. ед. | 3 | 3 |

5. Содержание дисциплины

5.1. Содержание разделов дисциплины

Раздел 1. Основные понятия и общие принципы построения вычислительных машин и вычислительных систем.

Тема 1.1. История вычислительной техники. Поколения ЭВМ.

Тема 1.2. Основные понятия и определения архитектуры ЭВМ. Принципы фон Неймана и классическая архитектура компьютера.

Тема 1.3. Функциональная и структурная организация ЭВМ. Информационно-логические основы построения ЭВМ.

Тема 1.4. Многоуровневая компьютерная организация. Цифровой логический уровень компьютера. Микроархитектурный уровень. CISC и RISC архитектура.

Тема 1.5. Уровень архитектуры команд. Структура и форматы машинных команд. Язык низкого уровня ассемблер. Инструкции. Операнды. Директивы. Трансляция и запуск программы.

Раздел 2. Функциональная и структурная организация центрального процессора ЭВМ

Тема 2.1. Назначение и структура центрального процессора. Назначение и организация устройства управления и арифметико-логического устройства. Микропроцессорная память. Регистры центрального процессора. Командный цикл процессора. Этапы исполнения команд процессором.

Тема 2.2. Производительность центрального процессора. Характеристики микропроцессора. Производство микропроцессора. Способы повышения производительности центрального процессора. Многоядерность.

Тема 2.3. Организация конвейерного режима работы процессора. Приостановка и опустошение конвейера. Суперскалярная архитектура.

Тема 2.4. Мультизадачный режим работы. Пакетный режим. Режим разделения времени. Режим реального времени. Аппарат прерываний. Привилегированный и ограниченный режимы работы процессора. Внутренние и внешние прерывания.

Раздел 3. Принципы организации системы памяти ВМ и ВС

Тема 3.1. Устройства хранения информации. Классификация устройств хранения информации. Иерархическая структура памяти компьютера. Физические принципы функционирования памяти.

Тема 3.2. Способы адресации информации в ВМ. Адреса памяти. Организация сегмента памяти. Работа со стековой памятью. Стековые регистры.

Тема 3.3. Принципы организации кэш-памяти. Принципы организации оперативной памяти. Внешняя память, физические принципы хранения информации. Статическая и динамическая память.

Тема 3.4. Понятие виртуальной памяти. Методы управления памятью: страничный, сегментный и странично-сегментный.

Тема 3.5. Файловая система. Задачи файловой системы. Имена файлов и индексные дескрипторы. Типы файлов. Права доступа к файлам. Файлы устройств и классификация устройств.

Раздел 4. Организация системного интерфейса и ввода-вывода информации

Тема 4.1. Устройства ввода и вывода информации. Организация ввода/вывода через пространство портов и через память. Прямой доступ к памяти периферийных устройств. Шины, их характеристики.

Тема 4.2. Классификация интерфейсов ввода-вывода. Параллельный и последовательный интерфейсы. Контроллеры внешних устройств. Драйверы устройств. Порты ввода/вывода.

Тема 4.3. Система и механизм прерываний микропроцессора. Виды прерываний. Аппаратные и программные прерывания. Управление прерываниями.

Тема 4.4. Организация архитектур с параллелизмом на уровне потоков и процессов (вычислительных систем). Классификация ВС. Основные компоненты ВС. Современные реализации мультимикропроцессоров, мультимикропроцессоров и кластеров.

5.2. Разделы дисциплин и виды занятий

Для направлений подготовки 38.03.05 «Бизнес-информатика», 09.03.03 «Прикладная информатика».

| № п/п | Наименование раздела дисциплины | Лекц. | Практ. зан. | Лаб. зан. | Се-мин. | СРС | Всего час. |
|--------------|---|--------------|--------------------|------------------|----------------|------------|-------------------|
| 1. | Основные понятия и общие принципы построения вычислительных машин и вычислительных систем | 4 | | 5 | | 19 | 28 |
| 2. | Функциональная и структурная организация центрального процессора ВМ | 4 | | 4 | | 18 | 26 |
| 3. | Принципы организации системы памяти ВМ | 4 | | 4 | | 18 | 26 |
| 4. | Организация системного интерфейса и ввода-вывода информации | 5 | | 4 | | 19 | 28 |
| Итого: | | 17 | | 17 | | 74 | 108 |

Для направлений подготовки 01.03.02 «Прикладная математика и информатика», 02.03.01 «Математика и компьютерные науки», 02.03.02 «Фундаментальная информатика и информационные технологии».

| № п/п | Наименование раздела дисциплины | Лекц. | Практ. зан. | Лаб. зан. | Се-мин. | СРС | Всего час. |
|--------|---|-------|-------------|-----------|---------|-----|------------|
| 1. | Основные понятия и общие принципы построения вычислительных машин и вычислительных систем | 4 | | 10 | | 14 | 28 |
| 2. | Функциональная и структурная организация центрального процессора ВМ | 4 | | 8 | | 14 | 26 |
| 3. | Принципы организации системы памяти ВМ | 4 | | 8 | | 14 | 26 |
| 4. | Организация системного интерфейса и ввода-вывода информации | 5 | | 8 | | 15 | 28 |
| Итого: | | 17 | | 34 | | 57 | 108 |

6. Лабораторный практикум

Раздел 1. Основные понятия и общие принципы построения вычислительных машин и вычислительных систем

Лабораторная работа 1. Основы интерфейса командной строки ОС GNU/Linux.

Лабораторная работа 2. Структура и процесс обработки программ на языке ассемблера NASM.

Раздел 2. Функциональная и структурная организация центрального процессора ВМ

Лабораторная работа 3. Трансляция программ на языке ассемблера. Системные вызовы в ОС GNU/Linux.

Лабораторная работа 4. Файл листинга. Программирование разветвляющегося процесса.

Раздел 3. Принципы организации системы памяти ВМ

Лабораторная работа 5. Отладчик GDB. Программирование цикла с переадресацией. Организация стека.

Лабораторная работа 6. Отладчик EDB. Понятие подпрограммы.

Раздел 4. Организация системного интерфейса и ввода-вывода информации

Лабораторная работа 7. Побитовые операции

Контроль знаний.

7. Практические занятия (семинары)

Практические занятия (семинары) не предусмотрены.

8. Материально-техническое обеспечение дисциплины

Дисплейные классы (ДК-3, ДК-4, ДК-6 расположенные по адресу: Москва, ул. Орджоникидзе, д. 3, корп. 5) с компьютерными рабочими местами пользователей с процессором не ниже Intel Core i3-550 3.2 GHz.

9. Информационное обеспечение дисциплины

а) Программное обеспечение: ОС Linux, Nasm, GDB, EDB.

б) Базы данных, информационно-справочные и поисковые системы: не требуются.

10. Учебно-методическое обеспечение дисциплины

а) Основная литература:

1. Таненбаум Э. Архитектура компьютера. 5-е изд. — СПб.: Питер, 2007. — 844 с.
2. Столяров А. В. Программирование на языке ассемблера NASM для ОС UNIX. — М.: МАКС Пресс, 2011. — 188 с.

б) Дополнительная литература:

1. Марек Р. Ассемблер на примерах. Базовый курс. — СПб: Наука и техника, 2005. — 240 с.
2. Жмакин А. П. Архитектура ЭВМ. — БХВ-Петербург, 2006. — 309 с.
3. Цилькер Б. Я., Орлов С. А. Организация ЭВМ и систем. — СПб.: Питер, 2006. — 667 с.
4. Гуров В. В., Чуканов В. О. Основы теории и организации ЭВМ. — М.: Интернет-университет информационных технологий; БИНОМ. Лаборатория знаний, 2006. — 272 с.

11. Методические указания для обучающихся по освоению дисциплины

Учебным планом на изучение дисциплины отводится один семестр. В дисциплине предусмотрены лекции, лабораторный практикум, контрольные мероприятия. В конце семестра проводится итоговый контроль знаний.

11.1. Методические указания по самостоятельному освоению теоретического материала по дисциплине

Лекционный материал дисциплины охватывает темы, указанные в разделе 5.1 программы дисциплины. В ТУИС (<http://esystem.pfur.ru>) по темам лекций размещены презентации. Рекомендуется по указанным темам в дополнение к презентациям изучить литературу, указанную в п. 10 программы дисциплины.

11.2. Методические указания по выполнению лабораторных работ

Задания по лабораторным работам выполняются индивидуально каждым студентом в дисплейных классах в соответствии с календарным планом и методическими указаниями по выполнению лабораторных работ по дисциплине. Часть лабораторных работ предусматривает задания для индивидуальной самостоятельной работы студента, обязательные для выполнения. Выполнение заданий для самостоятельной работы позволяет студенту приобрести дополнительные навыки и закрепить знания по изучаемой теме.

По результатам выполнения каждой лабораторной работы студентом готовится отчёт. Отчёты в электронном виде сдаются студентом на проверку через соответствующие разделы ТУИС (<http://esystem.pfur.ru>).

11.3. Методические указания по подготовке к контрольным мероприятиям

Контрольные мероприятия по дисциплине проводятся в форме тестирования в ТУИС (<http://esystem.pfur.ru>). Итоговый контроль в форме теста проводится по темам всех разделов дисциплины. Вопросы для подготовки к итоговому тестированию размещены в соответствующем разделе ТУИС (<http://esystem.pfur.ru>).

Паспорт фонда оценочных средств

| Код компетенции | Контролируемый раздел | Контролируемая тема | ФОСы | | | Баллы темы | Баллы раздела |
|---|--|--|-----------|-----------|--------------|------------|---------------|
| | | | Ауд. раб. | | Зач. | | |
| | | | ЛР | Тест | Итог. контр. | | |
| ОПК-3, ПК-30 (для 38.03.05), ОПК-1 (для 02.03.02 и 01.03.02), ОПК-4 (для 09.03.03), ОПК-2 (для 02.03.01), | Основные понятия и общие принципы построения ВМ и ВС | Основы интерфейса командной строки ОС GNU/Linux | 8 | 4 | 5 | 12 | 25 |
| | | Структура и процесс обработки программ на NASM | 8 | | | 13 | |
| | Функциональная и структурная организация ЦП ВМ | Трансляция программ на языке ассемблера. Системные вызовы в ОС GNU/Linux | 8 | 4 | 5 | 13 | 25 |
| | | Файл листинга. Программирование разветвляющегося процесса | 8 | | | 12 | |
| | Принципы организации системы памяти ВМ | Отладчик GDB. Программирование цикла с переадресацией. Организация стека | 12 | 4 | 5 | 16 | 33 |
| | | Отладчик EDB. Понятие подпрограммы | 12 | | | 17 | |
| | Организация системного интерфейса и I/O информации | Побитовые операции | 8 | 4 | 5 | 17 | 17 |
| Итого: | | | 64 | 16 | 20 | 100 | 100 |

Описание компетенций для направления 38.03.05:

ОПК-3 — способность работать с компьютером как средством управления информацией, работать с информацией из различных источников, в том числе в глобальных компьютерных сетях;

ПК-30 — владеть базовыми знаниями и навыками использования в профессиональной деятельности операционных систем, платформенных окружений, сетевых технологий.

Описание компетенций для направления 02.03.02:

ОПК-1 — способность использовать базовые знания естественных наук, математики и информатики, основные факты, концепции, принципы теорий, связанных с фундаментальной информатикой и информационными технологиями.

Описание компетенций для направления 09.03.03:

ОПК-4 — способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учётом основных требований информационной безопасности.

Описание компетенций для направления 01.03.02:

ОПК-1 — способность использовать базовые знания естественных наук, математики и информатики, основные факты, концепции, принципы теорий, связанных с прикладной математикой и информатикой.

Описание компетенций для направления 02.03.01:

ОПК-4 — способность решать стандартные задачи профессиональной деятельности на основе информационной и библиографической культуры с применением информационно-коммуникационных технологий и с учётом основных требований информационной безопасности.

Фонд оценочных средств

Балльно-рейтинговая система оценки уровня знаний

| Раздел | Тема | Формы контроля | | | Баллы темы | Баллы раздела |
|--|--|----------------|-----------|--------------|------------|---------------|
| | | Ауд. раб. | | Зач. | | |
| | | ЛР | Тест | Итог. контр. | | |
| Основные понятия и общие принципы построения ВМ и ВС | Основы интерфейса командной строки ОС GNU/Linux | 8 | 4 | 5 | 12 | 25 |
| | Структура и процесс обработки программ на NASM | 8 | | | 13 | |
| Функциональная и структурная организация ЦП ВМ | Трансляция программ на языке ассемблера. Системные вызовы в ОС GNU/Linux | 8 | 4 | 5 | 13 | 25 |
| | Файл листинга. Программирование разветвляющегося процесса | 8 | | | 12 | |
| Принципы организации системы памяти ВМ | Отладчик GDB. Программирование цикла с переадресацией. Орг. стека | 12 | 4 | 5 | 16 | 33 |
| | Отладчик EDB. Понятие подпрограммы | 12 | | | 17 | |
| Орг. сист. интерфейса и I/O информации | Побитовые операции | 8 | 4 | 5 | 17 | 17 |
| Итого: | | 64 | 16 | 20 | 100 | 100 |

Таблица соответствия баллов и оценок

| Баллы БРС | Традиционные оценки РФ | Оценки ECTS |
|-----------|------------------------|-------------|
| 95–100 | 5 | A |
| 86–94 | | B |
| 69–85 | 4 | C |
| 61–68 | 3 | D |
| 51–60 | | E |
| 31–50 | 2 | FX |
| 0–30 | | F |
| 51–100 | Зачёт | Passed |

Правила применения БРС

1. Раздел (тема) учебной дисциплины считается освоенным, если студент набрал более 50% от возможного числа баллов по этому разделу (теме).
2. Студент не может быть аттестован по дисциплине, если он не освоил все темы и разделы дисциплины, указанные в сводной оценочной таблице дисциплины.
3. По решению преподавателя и с согласия студентов, не освоивших отдельные разделы (темы) изучаемой дисциплины, в течение учебного семестра могут быть повторно проведены мероприятия текущего контроля успеваемости или выданы дополнительные учебные задания по этим темам или разделам. При этом студентам за данную работу засчитывается минимально возможный положительный балл (51% от максимального балла).
4. При выполнении студентом дополнительных учебных заданий или повторного прохождения мероприятий текущего контроля полученные им баллы засчитываются за конкретные темы. Итоговая сумма баллов не может превышать максимальное количество баллов, установленное по данным темам (в соответствии с приказом Ректора № 564 от 20.06.2013). По решению преподавателя предыдущие баллы, полученные студентом по учебным заданиям, могут быть аннулированы.
5. График проведения мероприятий текущего контроля успеваемости формируется в соответствии с календарным планом курса. Студенты обязаны сдавать все задания в сроки, установленные преподавателем.
6. Время, которое отводится студенту на выполнение мероприятий текущего контроля успеваемости, устанавливается преподавателем. По

завершении отведённого времени студент должен сдать работу преподавателю, вне зависимости от того, завершена она или нет.

7. Использование источников (в том числе конспектов лекций и лабораторных работ) во время выполнения контрольных мероприятий возможно только с разрешения преподавателя.
8. Отсрочка в прохождении мероприятий текущего контроля успеваемости считается уважительной только в случае болезни студента, что подтверждается наличием у него медицинской справки, заверенной круглой печатью в поликлинике № 25, предоставляемой преподавателю не позднее двух недель после выздоровления. В этом случае выполнение контрольных мероприятий осуществляется после выздоровления студента в срок, назначенный преподавателем. В противном случае отсутствие студента на контрольном мероприятии признается неуважительным.
9. Студент допускается к итоговому контролю знаний с любым количеством баллов, набранных в семестре.
10. Итоговый контроль знаний оценивается из 20 баллов, независимо от числа баллов за семестр.
11. Если в итоге за семестр студент получил менее 31 балла, то ему выставляется оценка F и студент должен повторить эту дисциплину в установленном порядке. Если же в итоге студент получил 31–50 баллов, т. е. FX, то студенту разрешается добор необходимого (до 51) количества баллов путём повторного одноразового выполнения предусмотренных контрольных мероприятий, при этом по усмотрению преподавателя аннулируются соответствующие предыдущие результаты. Ликвидация задолженностей проводится в период с 07.02 по 28.02 (с 07.09 по 28.09) по согласованию с деканатом.

Примерный перечень оценочных средств

| п/п | Наименование оценочного средства | Краткая характеристика оценочного средства | Представление оценочного средства в фонде |
|-------------------------------|---|--|---|
| Аудиторная работа | | | |
| 1. | Лабораторная работа | Система практических заданий, направленных на формирование практических навыков у обучающихся | Фонд практических заданий |
| 2. | Тест | Система стандартизированных заданий (вопросов), позволяющая автоматизировать процедуру измерения уровня знаний и умений обучающегося | База тестовых заданий |
| 3. | Зачёт | Форма проверки качества выполнения студентами лабораторных работ, домашних заданий и других заданий контрольных мероприятий в соответствии с утверждённой программой | Примеры заданий |
| Самостоятельная работа | | | |
| 1. | Подготовка отчётов по результатам выполнения лабораторных работ | Форма проверки качества выполнения студентами лабораторных работ в соответствии с утверждённой программой | Фонд практических заданий в рамках лабораторного практикума по дисциплине |

Учебным планом на изучение дисциплины отводится один семестр. В дисциплине предусмотрены лекции, лабораторный практикум, контрольные мероприятия. В конце семестра проводится итоговый контроль знаний.

Оценивание результатов освоения дисциплины производится в соответствии с балльно-рейтинговой системой. По дисциплине предусмотрен зачёт.

Итоговый контроль знаний по дисциплине проводится в форме тестирования, но при необходимости зачёт может проводиться в форме письменного ответа на вопросы из билетов.

Критерии оценки по дисциплине

95-100 баллов:

- полное и своевременное выполнение на высоком уровне лабораторных работ с оформлением отчётов, успешное прохождение контрольных мероприятий, предусмотренных программой курса;
- систематизированное, глубокое и полное освоение навыков и компетенций по всем разделам программы дисциплины;
- использование научной терминологии, стилистически грамотное, логически правильное изложение ответов на вопросы, умение делать обоснованные выводы;
- безупречное владение программным обеспечением, умение эффективно использовать его в постановке и решении научных и профессиональных задач;
- выраженная способность самостоятельно и творчески решать поставленные задачи;
- полная самостоятельность и творческий подход при изложении материала по программе дисциплины;
- полное и глубокое усвоение основной и дополнительной литературы, рекомендованной программой дисциплины и преподавателем.

86–94 балла:

- полное и своевременное выполнение на хорошем уровне лабораторных работ с оформлением отчётов, успешное прохождение контрольных мероприятий, предусмотренных программой курса;
- систематизированное, глубокое и полное освоение навыков и компетенций по всем разделам программы дисциплины;
- использование научной терминологии, стилистически грамотное, логически правильное изложение ответов на вопросы, умение делать обоснованные выводы;
- хорошее владение программным обеспечением, умение эффективно использовать его в постановке и решении научных и профессиональных задач;
- способность самостоятельно решать поставленные задачи в нестандартных производственных ситуациях;
- усвоение основной и дополнительной литературы, нормативных и законодательных актов, рекомендованных программой дисциплины и преподавателем.

69–85 баллов:

- своевременное выполнение на хорошем уровне лабораторных работ с оформлением отчётов, прохождение контрольных мероприятий, предусмотренных программой курса;
- хороший уровень культуры исполнения лабораторных работ;
- систематизированное и полное освоение навыков и компетенций по всем разделам программы дисциплины;
- владение программным обеспечением, умение использовать его в постановке и решении научных и профессиональных задач;

- способность самостоятельно решать проблемы в рамках программы дисциплины;
- усвоение основной литературы.

51-68 баллов:

- выполнение на удовлетворительном уровне лабораторных работ с оформлением отчётов, прохождение контрольных мероприятий, предусмотренных программой курса;
- систематизированное и полное освоение навыков и компетенций по всем разделам программы дисциплины;
- удовлетворительное владение программным обеспечением, умение использовать его в постановке и решении научных и профессиональных задач;
- способность решать проблемы в рамках программы дисциплины;
- удовлетворительное усвоение основной литературы;

31–50 баллов, НЕ ЗАЧТЕНО:

- невыполнение, несвоевременное выполнение или выполнение на неудовлетворительном уровне лабораторных работ, непрохождение контрольных мероприятий, предусмотренных программой курса;
- недостаточно полный объём навыков и компетенций в рамках программы дисциплины;
- неумение использовать в практической деятельности научной терминологии, изложение ответа на вопросы с существенными стилистическими и логическими ошибками;
- слабое владение программным обеспечением по разделам программы дисциплины, некомпетентность в решении стандартных (типовых) производственных задач;
- способность решать проблемы в рамках программы дисциплины;
- удовлетворительное усвоение основной литературы.

0-30 баллов, НЕ ЗАЧТЕНО:

- отсутствие умений, навыков, знаний и компетенций в рамках программы дисциплины;
- невыполнение лабораторных заданий, непрохождение контрольных мероприятий, предусмотренных программой курса; отказ от ответов по программе дисциплины;
- игнорирование занятий по дисциплине по неуважительной причине.

Комплект заданий для итогового контроля знаний

Итоговый контроль знаний по дисциплине проводится в форме компьютерного тестирования.

Примерный перечень вопросов итогового контроля знаний:

1. Что включает минимальная комплектация персонального компьютера?
2. Назовите устройства, входящие в состав процессора.
3. Когда были сформулированы основные идеи архитектуры ЭВМ?
4. Что такое арифметически-логическое устройство?
5. Что такое устройство управления?

6. Для чего служат регистры процессора?
7. Что такое тактовая частота процессора?
8. От чего зависит производительность работы компьютера (быстрота выполнения операций)?
9. В чём измеряется тактовая частота процессора?
10. Как называется организация работы ЦП, при которой выполнение следующей команды начинается до того, как закончится выполнение предыдущей?
11. Как называется организация работы ЦП, при которой два или более независимых процессора обрабатывают потоки команд из общей памяти?
12. Для чего служат регистры общего назначения?
13. Какие виды памяти относятся к внутренней памяти ПК?
14. Что означает адресуемость оперативной памяти?
15. В чём заключается назначение внешней памяти компьютера?
16. Для чего служит постоянное запоминающее устройство ПК?
17. Какое из устройств предназначено для ввода информации?
18. Какой принцип записи и считывания информации используется в лазерном диске?
19. Какая инструкция NASM дублирует слово, расположенное по адресу `esi`, в регистр `eax`?
20. Какое действие произойдёт при выполнении инструкции `mov eax, esi`?
21. В каком регистре при использовании инструкции `loop` задаётся количество повторений цикла?
22. Какая структура данных называется стеком?
23. Какая инструкция используется для добавления элемента на вершину стека в языке ассемблера?
24. Какая инструкция используется для извлечения элемента с вершины стека в языке ассемблера?
25. Какая инструкция используется для работы с подпрограммами в основной программе в языке ассемблера?
26. Какой инструкцией можно при написании программы заменить инструкцию `mov ecx, 0`?
27. Какой инструкцией можно при написании программы заменить инструкцию `div ecx, 2`?
28. Вызов какой инструкции выполняет системный вызов функции ядра Linux?
29. Какой номер системного вызова используется ядром Linux для выполнения системной функции чтения (`sys_read`)?
30. Какой номер системного вызова используется ядром Linux для выполнения системной функции записи (`sys_write`)?
31. Что не используется для объявления инициализированного пространства для хранения данных?
32. Для чего используется директива `EQU`?
33. С помощью какой команды можно выполнить N инструкций в GDB?
34. Что входит в структуру файла листинга?
35. Какое расширение у Makefile?
36. После выполнения команды

- ```
nasm -f elf64 lab.asm -o bal.o -l alb.lst -g
```
- сколько файлов будет находиться в текущей папке?
37. Что означает следующая ошибка в nasm: symbol «message» undefined?
  38. Какие регистры используются для хранения индексов при работе с массивами?
  39. Какие регистры используются для работы со стеком?
  40. Какая инструкция возвращает управление вызывающей программе?
  41. Для чего используются квадратные скобки в инструкциях NASM?
  42. Для чего используются следующие компоненты NASM: db, dw, dd, dq и dt?
  43. Может ли быть позднее переопределено msglen, записанное в следующем виде: msglen: equ \$-message?
  44. Какой режим отображения синтаксиса машинных команд используется по умолчанию в EDB?
  45. Для чего используется операция idiv?
  46. Какое значение содержит флаг CF при выполнении команд сдвига?
  47. Для чего используется команда not?
  48. В какой секции пишется код программы в NASM?

## **Критерии оценки итогового тестирования**

Итоговое тестирование оценивается в соответствии с БРС и паспортом ФОС. Проверяется правильность ответов на вопросы теста.

## **Комплект разноуровневых задач (заданий)**

### **1. Задания репродуктивного уровня**

В качестве заданий репродуктивного уровня предлагаются вопросы для самопроверки и обсуждения по темам курса (см. лабораторный практикум).

### **2. Задания реконструктивного уровня**

В качестве заданий реконструктивного уровня предполагаются задания лабораторного практикума.

## **Критерии оценки выполнения заданий по лабораторным работам**

Оцениваются полнота выполнения работы, оформление результатов, полнота ответов на контрольные вопросы, если это предусмотрено заданием.

## Сведения об авторах

Кулябов Дмитрий Сергеевич — доцент, доктор физико-математических наук, профессор кафедры прикладной информатики и теории вероятностей РУДН.

Королькова Анна Владиславовна — доцент, кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Демидова Анастасия Вячеславовна — кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Геворкян Мигран Нельсонович — кандидат физико-математических наук, доцент кафедры прикладной информатики и теории вероятностей РУДН.

Велиева Татьяна Рефатовна — ассистент кафедры прикладной информатики и теории вероятностей РУДН.

Учебное издание

**Анастасия Вячеславовна Демидова,  
Татьяна Рефатовна Велиева,  
Анна Владиславовна Королькова,  
Мигран Нельсонович Геворкян,  
Дмитрий Сергеевич Кулябов**

## **Архитектура вычислительных систем**

Редактор *И. Л. Панкратова*  
Технический редактор *Н. А. Ясько*  
Компьютерная вёрстка *А. В. Королькова, Д. С. Кулябов*

Подписано в печать 17.12.2018 г. Формат 60×84/16. Печать офсетная.  
Усл. печ. л. 5,5. Тираж 500 экз. Заказ № 1520.

---

Российский университет дружбы народов  
115419, ГСП-1, г. Москва, ул. Орджоникидзе, д. 3

---

Типография РУДН  
115419, ГСП-1, г. Москва, ул. Орджоникидзе, д. 3, тел. 952-04-41