Московский государственный технический университет
имени Н.Э. Баумана

==========

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

Г.С. Иванова, Т.Н. Ничушкина

ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА АССЕМБЛЕРЕ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Учебно-методическое пособие

Москва

УДК 004.432

ББК 32.973.26-018.1

И 21

Иванова Г.С., Ничушкина Т.Н.

Лабораторный практикум по программирование на ассемблере в операционной системе LINUX. М.: МГТУ имени Н.Э. Баумана, 2022. 77 с.

Издание содержит методический материал для выполнения практикума из пяти лабораторных работ по дисциплине «Машинно-зависимые языки и основы компиляции». Четыре лабораторные посвящены первичному знакомству с программированием на ассемблере *NASM* в операционной системе *LINUX*, а пятая работа предполагает изучение особенностей и формирование умений и навыков создания программ, включающих модули, написанные на разных языках программирования, включая ассемблер.

Определены цели лабораторных работ, пояснены сложные моменты, приведены задания, предложен порядок их выполнения и сформулированы требования к отчету. Также даны рекомендации по настройке транслятора *nasm*, компоновщика *ld* и отладчика *edb*. Рассмотрены форматы представления данных и способы их адресации. Приведены фрагменты программ, демонстрирующие особенности программирования на ассемблере.

Для студентов МГТУ имени Н.Э. Баумана, обучающихся по программам бакалавриата направлений «Информатика и вычислительная техника» и «Прикладная информатика».

Рекомендовано Научно-методическим советом МГТУ им. Н.Э. Баумана в качестве учебного пособия

Учебное издание

Иванова Галина Сергеевна

Ничушкина Татьяна Николаевна

Лабораторный практикум по программированию на ассемблере в операционной системе LINUX.

Оглавление

Предисловие

Учебно-методическое пособие составлено в соответствии с самостоятельно устанавливаемыми образовательными стандартами (СУОС), основными образовательными программами по направлениям подготовки бакалавров 09.03.01 «Информатика и вычислительная техника» и 09.03.03 «Прикладная информатика», а также программой дисциплины «Машинно-зависимые языки и основы компиляции». Издание предназначено для изучения средств программирования на языке ассемблера при выполнении лабораторных работ всех модулей указанной дисциплины.

Цель учебно-методического пособия – закрепление теоретических знаний и формирование практических навыков, необходимых для разработки программ на языке ассемблера, формирование умений создания, тестирования и отладки программ на машинно-зависимых языках.

Задача издания – предоставить студентам материал, который будет способствовать более качественному усвоению материала всех модулей дисциплины. Для успешного усвоения материала необходимо внимательно прочитать текст издания и выполнить все предлагаемые задания. Ответы на вопросы позволят обучающемуся оценить степень понимания и усвоения теоретических положений.

В целом лабораторный практикум по дисциплине обеспечивает базовые знания для формирования следующих собственных общепрофессиональных и профессиональных компетенций:

– способен, используя эффективные подходы и средства, разрабатывать алгоритмы и программы, пригодные для практического применения;

– способен выполнять работы по созданию и модификации программных или программно-аппаратных компонентов ИТ-систем цифровой экономики.

Введение

Язык ассемблера или ассемблер (англ. assembly language) — язык низкого уровня с командами, обычно соответствующими командам процессора (так называемым машинным командам). Это соответствие позволяет отнести язык к группе машинно-зависимых, к которой относятся также машинные языки.

В отличие от машинных языков в языках ассемблера используются мнемонические (буквенные) обозначения команд, а также символическая адресация для команд и данных. Это существенно облегчает процесс написания программ, непосредственно преобразуемых в машинные коды.

Кроме того, языки ассемблера обычно включают макрокоманды, которым при переводе в машинный язык соответствует сразу некоторая настраиваемая последовательность машинных команд, что сокращает размер программы, позволяя не расписывать цепочки машинных команд, реализующие часто встречающиеся действия.

Изучение основ программирования на языке ассемблера является необходимой частью обучения специалистов в области проектирования, как аппаратных, так и программных средств вычислительной технике, поскольку позволяет отчетливо понимать, как работает вычислительная машина в процессе выполнения программы.

При выполнении лабораторных работ по дисциплине используется следующее программное обеспечение, работающее под управлением операционной системы *Linux Astra*:

- текстовые редакторы *Kate* и *GHex*;
- ассемблер nasm и компоновщик *Ld*;
- отладчик *Edb*;
- среды программирования *Lazarus* и *Qt Creator*.

Все указанное выше программное обеспечение является свободным.

1. Лабораторная работа Изучение среды и отладчика ассемблера

Цель работы: изучение процессов создания, запуска и отладки программ на ассемблере *Nasm* под управлением операционной системы *Linux*, а также особенностей описания и внутреннего представления данных.

Объем работы: 2 часа.

1.1. Описание программного обеспечения

1.1.1. Структура простейшей программы на ассемблере NASM

Nasm – свободный (лицензии *LGPL* и *BSD*) ассемблер для написания 32-х и 64-х разрядных программ для процессоров *Intel x*86. Транслятор поддерживает *Intel*-формат синтаксиса языка.

Лабораторная работа предполагает работу либо с 32-х разрядными, либо с 64-х разрядными программами на ассемблере. Выбор типа программы зависит от конфигурации используемых технических средств и, соответственно размерности операционной системы. Также допустимо на 64-разрядном компьютере работать с 32-х разрядными программами в режиме эмуляции.

И 32-х и 64-х разрядные программы состоят из трех секций (сегментов):

.text – сегмент кода;

.data – сегмент инициализированных данных;

.bss – сегмент неинициализированных данных.

Заготовка программы 32-х разрядной конфигурации выгладит следующим образом:

```
    section .data  ; сегмент инициализированных переменных
ExitMsg db "Press Enter to Exit",10 ; выводимое сообщение
lenExit equ $-ExitMsg

    section .bss  ; сегмент неинициализированных переменных
InBuf   resb    10   ; буфер для вводимой строки
lenIn   equ     $-InBuf
```

```
    section .text    ; сегмент кода
    global  _start
_start:
    ; write
    mov     eax, 4      ; системная функция 4 (write)
    mov     ebx, 1          ; дескриптор файла stdout=1
    mov     ecx, ExitMsg    ; адрес выводимой строки
    mov     edx, lenExit    ; длина выводимой строки
    int     80h             ; вызов системной функции
    ; read
    mov     eax, 3          ; системная функция 3 (read)
    mov     ebx, 0          ; дескриптор файла stdin=0
    mov     ecx, InBuf      ; адрес буфера ввода
    mov     edx, lenIn      ; размер буфера
    int     80h             ; вызов системной функции
    ; exit
    mov     eax, 1          ; системная функция 1 (exit)
    xor     ebx, ebx        ; код возврата 0
    int     80h             ; вызов системной функции
```

Заготовка включает коды обращения к системным функциям для выполнения операций ввода с клавиатуры, вывода на экран и завершения программы.

Заготовка 64-х разрядной программы содержит те же операции, но использует для их вызова другие номера системных функций и регистры. Кроме того, для вызова функций операционной системы вместо прерывания *int* 80h в этом случае обычно используют команду системного вызова *syscall*.

Заготовка 64-разрядной программы на ассемблере:

```
    section .data    ; сегмент инициализированных переменных
ExitMsg db "Press Enter to Exit",10 ; выводимое сообщение
```

```
lenExit equ $-ExitMsg
   section .bss      ; сегмент неинициализированных переменных
InBuf   resb    10  ; буфер для вводимой строки
lenIn   equ     $-InBuf
   section .text  ; сегмент кода
   global  _start
_start:
    ; write
    mov     rax, 1          ; системная функция 1 (write)
    mov     rdi, 1          ; дескриптор файла stdout=1
    mov     rsi, ExitMsg    ; адрес выводимой строки
    mov     rdx, lenExit    ; длина строки
    syscall                 ; вызов системной функции
    ; read
    mov     rax, 0          ; системная функция 0 (read)
    mov     rdi, 0          ; дескриптор файла stdin=0
    mov     rsi, InBuf      ; адрес вводимой строки
    mov     rdx, lenIn      ; длина строки
    syscall                 ; вызов системной функции
    ; exit
    mov     rax, 60         ; системная функция 60 (exit)
    xor     rdi, rdi        ; return code 0
    syscall                 ; вызов системной функции
```

Эти заготовки будут использоваться как исходные тексты в 4-х первых лабораторных и первом домашнем задании.

1.1.2. Ввод, трансляция, компоновка и выполнение программы

Предложенные в предыдущем разделе заготовки программ можно выполнить. Для этого их необходимо ввести в компьютер, преобразовать в

машинный код и скомпоновать со всеми требуемыми подпрограммами. Первую операцию выполняет текстовый редактор, вторую – транслятор, третью – компоновщик. И только после этого полученную выполняемую программу можно запустить на выполнение.

Для ввода программы в компьютер используют текстовый редактор, который не осуществляет никакой дополнительной разметки текста, например текстовый редактор *Kate*. Для подготовки программы к выполнению вызывают транслятор *nasm* и компоновщик *ld*. В результате работы транслятор создает объектный модуль, которые затем подается на вход компоновщика, формирующего исполняемую программу (рис. 1).

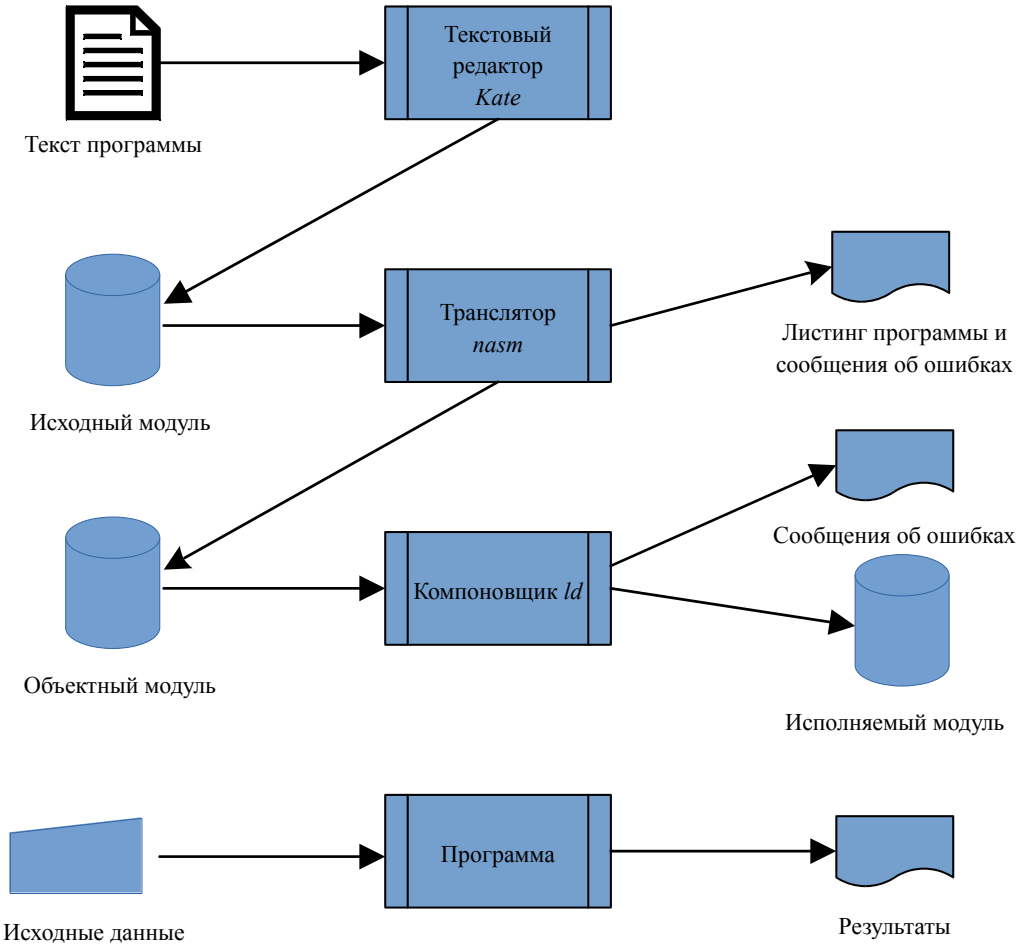

Рис. 1. Схема процесса формирования исполняемого модуля на ассемблере

Полученный исполняемый модуль можно выполнить непосредственно или загрузить в отладчик для поиска ошибок.

Трансляцию и компоновку программы осуществляют в консольном режиме с использованием эмулятора терминала *fly*. Для вызова терминала используют пункт Системные/Терминал *Fly* меню Пуск.

При вводе команд трансляции/компоновки/вызова отладчика и др. в окне терминала следует соблюдать правила указания имен файлов в командах консольного режима.

Файловая система и указание имен файлов в командах консольного режима *Linux*. Для того чтобы понять, как именуются файлы в операционных системах типа *Linux*, необходимо изучить структуру файловой системы операционной системы (рис. 2).

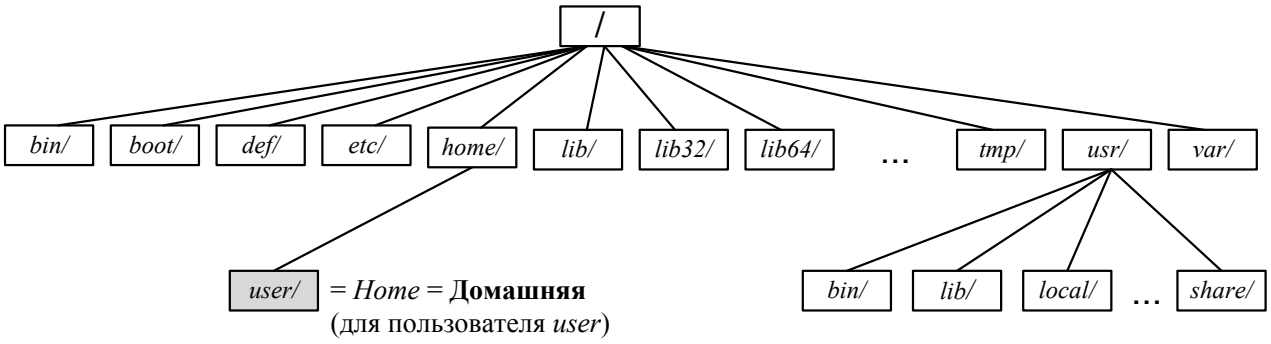

Рис. 2. Структура файловой системы операционных систем типа *Linux*

Особенностью систем рассматриваемого типа является то, что пользователь, вошедший в систему, оказывается в папке Домашняя, которая у каждого пользователя системы своя. Эти папки в системе названы по имени пользователей и расположены в папке *home* операционной системы, но отображаются в дереве файлов именно как Домашняя.

Только в папке Домашняя и вложенных в нее папках пользователь может без ограничений создавать, корректировать и удалять папки и файлы. Для изменения информации в остальных (системных) папках ему необходимы права администратора. Однако содержимое системных папок можно посмотреть в Менеджере файлов, используя ссылки Компьютер на рабочем столе системы или Менеджер файлов на панели быстрого запуска (нижняя строка).

Также следует иметь в виду, что файлы и папки, имена которых начинаются с точки, считаются скрытыми и их можно сделать невидимыми, применив соответствующую настройку Менеджера файлов.

Как сказано выше текущим каталогом (папкой) при входе пользователя в систему является папка Домашняя (*/home/user*). Для обращения к файлам текущей папки можно использовать короткие имена, не включающие путь, например **`file.dat`**.

При обращении к файлам, расположенным не в текущем каталоге, следует указывать либо полный путь к нему по правилам *Linux* – абсолютное имя файла, либо путь относительно текущего каталога – относительное имя файла, например:

`/home/user/prog1.asm` – абсолютное имя файла, где «/» – корневой каталог *Linux*;

`~/progs/prog1.asm` – относительное имя файла, где «~/» подразумевает домашний каталог */home/user* пользователя *user Linux Astra*;

`../prog1.asm` – относительное имя файла, где «../» означает родительский для текущего каталог.

Для выполнения пользовательской программы из текущего каталога перед ней указывают символы обозначения текущего каталога «./», например **`./prog1`**.

Проблем с вызовом большинства установленных в другие каталоги программ обычно не возникает: они чаще всего находятся в каталогах, поиск в которых предполагается по умолчанию («в путях автовызова»): их вызывают также по короткому имени, например **`nasm`**. Однако, если программа находится не в путях автовызова, то ее следует вызывать используя полное абсолютное или относительное имя.

Примечание. Далее для описания терминальных команд используется нотация, в которой необязательные элементы указываются в квадратных скобках, а альтернативные варианты – в фигурных.

Трансляция программы. Вызов транслятора *nasm* (расположенного в путях автовызова) в обычном варианте имеет следующий вид:

```
nasm -f <Формат> <Имя_исходного_модуля>
     [-o <Имя_объектного_модуля>]
     [-l <Имя_файла_листинга>]
```

где

<Формат> — формат объектного модуля — *elf32* (*elf*) или *elf64* для 32-х и 64-х разрядных *Linux*-программ соответственно;

<Имя_исходного_модуля> — имя файла с суффиксом *.asm* с текстом программы на ассемблере;

<Имя_объектного_модуля> — имя выходного файла, содержащего машинный код программы и служебные таблицы для дальнейшей компоновки; согласно описанию команды выше это имя можно не указывать, тогда по умолчанию ассемблер сформирует файл с тем же именем и суффиксом *.o*;

<Имя_файла_листинга> — имя файла, в который транслятор должен записать листинг программы, включая, если они есть, сообщения об ошибках; если опция *-l* и имя файла не указаны, то листинг программы не создается.

Если транслятор обнаруживает ошибки, то информация о них с указанием номеров строк, в которых обнаружены ошибки, кроме листинга выводится также в окно терминала после команды вызова транслятора. Файл объектного модуля при этом не создается.

Компоновка программы. При компоновке программы необходимо учитывать реальную разрядность компьютера и операционной системы. В случае если 32-х разрядную программу необходимо выполнить на 64-х разрядном оборудовании, следует при компоновке заказывать эмуляцию 32-х разрядной адресации для 64-х разрядной операционной системы.

Вызов компоновщика в простейших случаях выглядит следующим образом:

```
ld [-m elf_i386] [-o <Имя_исполняемого_модуля>]
    <Имя_объектного_модуля>
```

где

elf_i386 — режим эмуляции; опцию указывают, если компонуется 32-х разрядная программа для выполнения на 64-х разрядном компьютере;

<Имя_исполняемого_модуля> — имя файла результирующей программы; если опция не указана, то строится программа с именем *a.out*;

<Имя_объектного_модуля> — имя файла, содержащего машинный код программы.

Сообщение об ошибках сборки программы компоновщик выводит в окно терминала. Если ошибки отсутствуют, то никаких сообщений не выдается.

Выполнение программы. Запуск программы на выполнение (программа в текущем каталоге) осуществляется по ее имени:

```
./<Имя_исполняемого_модуля>
```

Программа при запуске загружается в оперативную память и начинает выполняться.

Выполнение программы в отладчике. В настоящей лабораторной работе предусматривается изучение отладчика *edb* с графическим интерфейсом. Запуск отладчика осуществляется по имени в окне терминала *fly*:

```
ebd
```

После ввода команды на экране появится окно отладчика без программы. Для открытия в отладчике программы на ассемблере следует использовать пункт меню отладчика *File*/*Open*. После выбора программы в окне отладчика появляются следующие данные (рис. 3):

- машинный и дисассемблерованный коды программы (вверху слева),
- содержимое регистров (вверху справа),
- шестнадцатеричный дамб сегментов данных (внизу слева),
- шестнадцатеричный дамб стека (внизу справа).

Отладчик позволяет выполнять программу по шагам, заходя или не заходя в подпрограммы, и при этом анализировать изменения содержимого регистров, памяти и стека. Для этого используют соответствующие пункты меню или следующие клавиши:

- *F*7 — выполнить шаг с заходом в подпрограмму;
- *F*8 — выполнить шаг без захода в подпрограмму;
- *Ctrl* + *F*2 — начать отладку сначала;
- *Ctrl* + *F*9 — выполнить подпрограмму до возврата из нее.

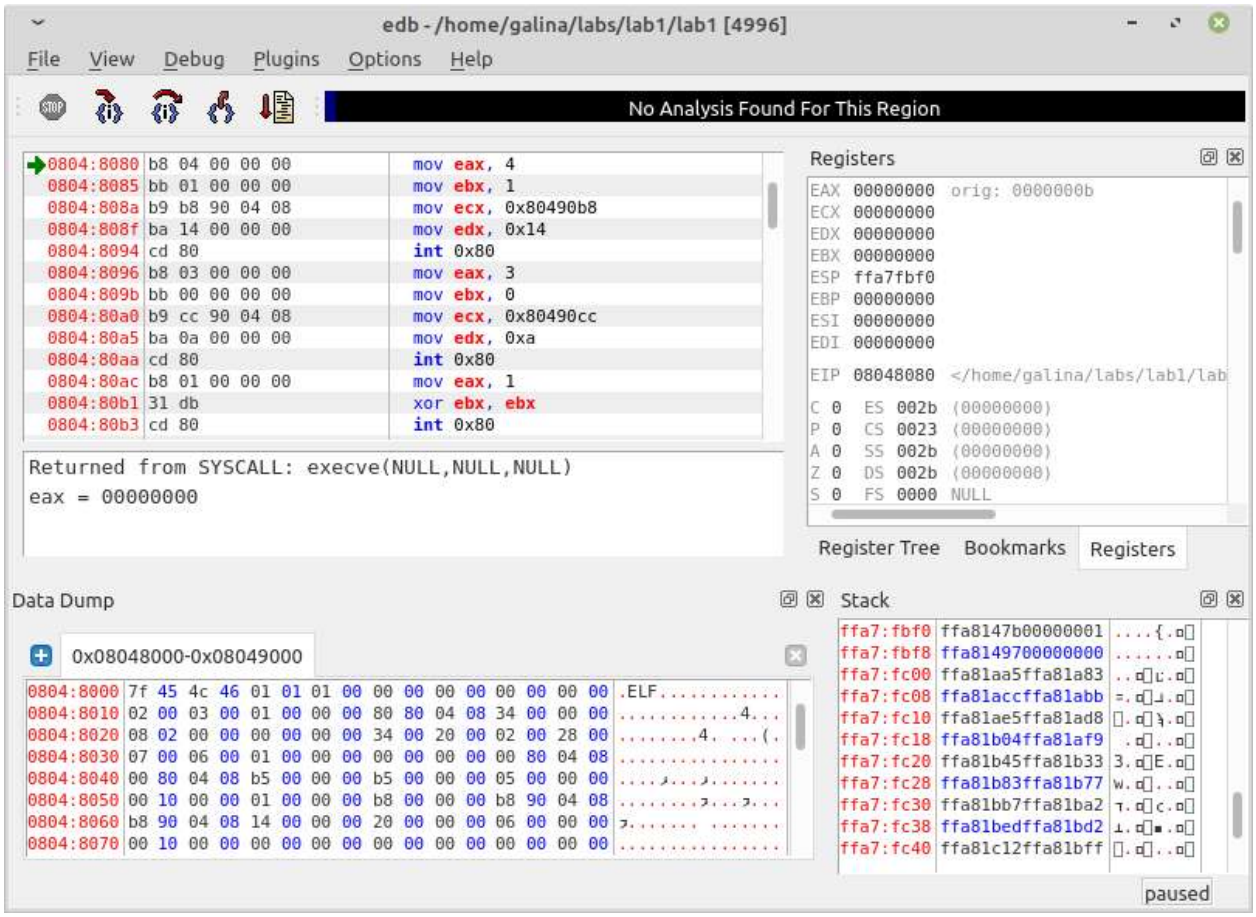

Рис. 3. Графический интерфейс отладчика *edb*

Для настройки на нужный адрес области сегмента данных следует вызвать контекстное меню щелчком правой клавиши мыши в области панели и выбрать команду *Goto Exrpession*. В появившемся окне в поле адреса надо ввести адрес в шестнадцатеричном формате, используя обычное обозначение адреса в языке программирования *C++*, например 0x08049000.

Адрес начала сегмента кода зависит от разрядности и версии *Linux*. Сегмент инициированных данных начинается после завершения сегмента кода. Каждое значение занимает столько байт, сколько резервируется используемой для описания данных директивой. В отладчике каждый байт представлен двумя шестнадцатеричными цифрами. Кроме того, используется обратный порядок байтов, т.е. младший байт числа находится в младших адресах памяти (перед старшим). Если шестнадцатеричная комбинация соответствует коду символа, то он высвечивается в соседнем столбце, иначе в нем высвечивается точка.

Неинициированные данные располагаются после инициированных. При этом возможно выравнивание на заданную границу: адрес, кратный восьми, шестнадцати и т.п., т.е. может быть пропущено несколько байтов.

При выполнении команды в отладчике можно наблюдать изменение данных в памяти и/или в регистрах, отслеживая процесс выполнения программы и контролируя правильность промежуточных результатов.

Так после выполнения первой команды число *A* скопируется в регистр *EAX*, который при этом подсвечивается красным. При записи в регистр порядок байт меняется на прямой, при котором первым записан старший байт.

1.1.3. Описание данных в программе на ассемблере

Все данные (исходные, промежуточные и результаты), используемые в программах на ассемблере, обязательно должны быть объявлены. При этом для данных задается объем выделяемой памяти в байтах.

Директива объявления данных имеет следующий формат:

[<Имя>] [times <Константа>]<Директива>[<Список_инициализаторов>]

где <Имя> – имя поля данных, которое может не присваиваться;

times <Константа> — псевдо-инструкция повторения полей данных, константа определяет количество повторений;

<Директива> — команда, объявляющая тип описываемых данных:

- *db* — определить байт,
- *dw* — определить слово (2 байта),
- *dd* — определить двойное слово (4 байта),
- *dq* — определить счетверенное слово (8 байт),
- *dt* — определить 10 байт;

<Список_инициализаторов> — последовательность инициализирующих констант, указанных через запятую.

В качестве инициализаторов при описании данных применяются:

- целые константы [<Знак>]<Целое> [<Основание системы счисления>], например:
  - -43236 – целые десятичные числа,
  - 0*x*22, 23*h*, $0*AD* – целые шестнадцатеричные числа (если шестнадцатеричная константа начинается с буквы, то перед ней указывается 0),
  - 0111010*b* – целое двоичное число;
- вещественные числа [<Знак>] <Целое>.[Целое][*e*[<Знак>] [<Целое>], например: -2., 34.*e*-28;
- символьные константы в апострофах или кавычках, например: '*A*' или "*A*";
- строковые константы в апострофах или кавычках, например, '*ABCD*' или "*ABCD*".

1.2. Порядок выполнения работы

1.2.1. Для хранения всех программ лабораторных работ на компьютере в операционной системе *Linux Astra* создайте специальные каталог и подкаталог. Каталог *labs* целесообразно разместить в подкаталоге Домашний (*/home/user*) пользователя *user* системы *Linux*.

Создать каталог и подкаталог можно в эмуляторе терминала *fly*, используя команду *mkdir*:

```
...$ mkdir labs
...$ mkdir labs/lab1
```

где `...$` = `(base)user@astra:~$` – приглашение командной строки терминала.

1.2.2. Объявите подкаталог *labs*/*lab1*текущим:

```
...$ cd labs/lab1
```

При этом приглашение в командной строке изменится, показывая, что текущим каталогом стал каталог */home/user/labs/lab*1:

```
(base)user@astra:~/labs/lab1$
```

1.2.3. Используя меню Пуск, вызовите текстовый редактор *Kate* (раздел Офис) и введите заготовку 32-х или 64-х разрядной программы на ассемблере. Сохраните программу с именем *lab1.asm* в подкаталоге *labs/lab1*.

Внимательно изучите структуру программы и способ программирования системных вызовов для выполнения операций ввода-вывода и завершения программы. Зафиксируйте текст программы с комментариями в отчете.

1.2.4. Выполните трансляцию программы с листингом. Для этого

- для 32-х разрядной программы следует ввести команду:

```
… :~/labs/lab1$ nasm -f elf lab1.asm -l lab1.lst
```

- для 64-х разрядной:

```
… :~/labs/lab1$ nasm -f elf64 lab1.asm -l lab1.lst
```

В результате вы должны получить объектный модуль *lab1.o* и файл листинга *lab1.lst*. Убедитесь, что операция прошла без ошибок.

1.2.5. Для компоновки 32-х или 64-х разрядной программы, которая будет выполняться на компьютере той же размерности, следует ввести:

```
… :~/labs/lab1 $ ld -o lab1 lab1.o
```

Для компоновки 32-разрядной программы, которая будет выполняться на 64-х разрядном компьютере, необходимо запросить режим эмуляции:

```
… :~/labs/lab1 $ ld -m elf_i386 -o lab1 lab1.o
```

После команды в окне терминала, если они есть, выводятся сообщения об ошибках. Если все прошло без ошибок, то в том же каталоге появится файл исполняемой программы *lab1*.

1.2.6. Запустите программу на выполнение:

```
… :~/labs/lab1 $ ./lab1
```

Запущенная программа должна вывести текст:

```
Press Enter to Exit
```

и ожидать нажатия клавиши *Enter*. После нажатия клавиши *Enter* выполнение программы завершится.

Следует иметь в виду, что буфер ввода предусматривает ввод до 10 символов в то время как при нажатии клавиши *Enter* в буфер записывается единственный символ с кодом 10_{10} =$0A_{16}$. Символ с кодом 13_{10}, заносимый в буфер ввода в операционных системах семейства *Windows*, в операционных системах типа *Linux* при нажатии на клавишу *Enter* в буфер не заносится.

1.2.7. Запустите отладчик *edb*. Для этого следует в окне терминала ввести команду:

```
… :~/labs/lab1 $ edb
```

Средствами графического интерфейса отладчика откройте в нем исполняемую программу *lab1* и проанализируйте, что вы видите в его окне. Найдите машинное представление программы, ее дисассемблированный код, содержимое регистров и т.д. Выполните программу по шагам, контролируя содержимое регистров и оперативной памяти.

1.2.8. Для изучения возможностей отладчика добавьте в заготовку несколько команд для вычисления результата следующего выражения:

$X=A+5-B$

Данные для программы задайте константами, поместив их описание в раздел инициированных данных *.data*:

```
A           DWORD   -30
B           DWORD   21
```

Для результата вычислений – переменной *X* – необходимо зарезервировать место, поместив описание соответствующей неинициализированной переменной в раздел неинициализированных данных *.bss*:

```
X          resd  1
```

Фрагмент кода программы, выполняющей сложение и вычитание, поместите в сегмент кодов после метки *start*:

```
start:  mov     EAX,[A] ;  загрузить число A в регистр EAX
        add     EAX,5   ;  сложить EAX и 5, результат в EAX
        sub     EAX,[B] ;  вычесть число B, результат в EAX
        mov     [X],EAX ;  сохранить результат в памяти
        …
```

Сохраните программу с тем же именем, затем выполните ее трансляцию, компоновку и загрузку в отладчик. Зафиксируйте изменение программы в отчете.

1.2.9. Найдите в отладчике внутреннее представление исходных данных, отразите его в отчете и поясните.

Проследите в отладчике выполнение программы и зафиксируйте в отчете результаты выполнения каждой добавленной команды (изменение регистров, флагов и полей данных).

1.2.10. Введите следующие строки в разделы описания инициированных и неинициализированных данных и определите с помощью отладчика внутренние представление этих данных в памяти. Результаты проанализируйте и занесите в отчет.

```
val1    db   255
chart   dw   256
lue3    dw  -128
v5      db   10h
```

```
        db   100101B
beta    db   23,23h,0ch
sdk     db   "Hello",10
min     dw  -32767
ar      dd  12345678h
valar times 5 db  8
alu     resw   10
f1      resb   5
```

1.2.11. Определите в памяти следующие данные:

а) целое число 25 размером 2 байта со знаком;

б) двойное слово, содержащее число -35;

в) символьную строку, содержащую ваше имя (русскими буквами и латинскими буквами).

Зафиксируйте в отчете описание и внутреннее представление этих данных и дайте пояснение.

1.2.12. Определите несколькими способами в программе числа, которые во внутреннем представлении (в отладчике) будут выглядеть как **25 00 *и* 00 25**. Проверьте правильность ваших предположений, введя соответствующие строки в программу. Зафиксируйте результаты в отчете.

1.2.13. Добавьте в программу переменную *F1*=65535 размером слово и переменную *F2*= 65535 размером двойное слово. Вставьте в программу команды сложения этих чисел с 1:

```
        add    [F1],1
        add    [F2],1
```

Проанализируйте и прокомментируйте в отчете полученный результат (обратите внимание на флаги).

1.3. Требования к отчету

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш не допускается). Схемы и таблицы также должны быть напечатаны при помощи компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Содержание отчета по лабораторной работе № 1 указано в тексте задания на работу.

Отчет должны иметь титульный лист, на котором указывается:

а) наименование факультета и кафедры;

б) название дисциплины;

в) номер и тема лабораторной работы;

г) фамилия преподавателя, ведущего занятия;

д) фамилия, имя и номер группы студента.

1.4. Требования к защите

Защита лабораторной предполагает ответы на вопросы преподавателя. При этом работа считается защищенной, если студент правильно и уверенно отвечает на 2-3 вопроса преподавателя, может указать в программе те или иные операции, а также поясняет работу указанного фрагмента.

Контрольные вопросы

1. Дайте определение ассемблеру. К какой группе языков он относится?

2. Из каких частей состоит заготовка программы на ассемблере?

3. Как запустить программу на ассемблере на выполнение? Что происходит с программой на каждом этапе обработки?

4. Назовите основные режимы работы отладчика. Как осуществить пошаговое выполнение программы и просмотреть результаты выполнения машинных команд.

5. В каком виде отладчик показывает положительные и отрицательные целые числа? Как будут представлены в памяти числа:

```
A    dw 5,-5 ?
```

Как те же числа будут выглядеть после загрузки в регистр *AX*?

6. Каким образом в ассемблере программируются выражения? Составьте фрагмент программы для вычисления $C=A+B$, где A, B и C – целые числа формата *BYTE*.

2. Лабораторная работа Программирование целочисленных вычислений

Цель работы: изучение форматов машинных команд, команд целочисленной арифметики ассемблера и программирование целочисленных вычислений.

Объем работы: 2 часа.

Пример задания. Разработать программу на языке ассемблера, вычисляющую выражение:

$x := g^2 - 4(k-23)/(g+k)$.

Варианты заданий приведены на странице дисциплины на сайте кафедры.

2.1. Описание используемого программного обеспечения

2.1.1. Форматы машинных команд *IA*-32

Структура команд ассемблера определяется форматом соответствующих машинных команд, поэтому очень важно разобраться, какие компоненты входят в машинную команду и какие способы адресации данных можно использовать при написании программы на ассемблере.

Размер машинной команды процессора *IA*-32 может меняться от 1 до 15 байт. Команда имеет следующую структуру (рис. 4):

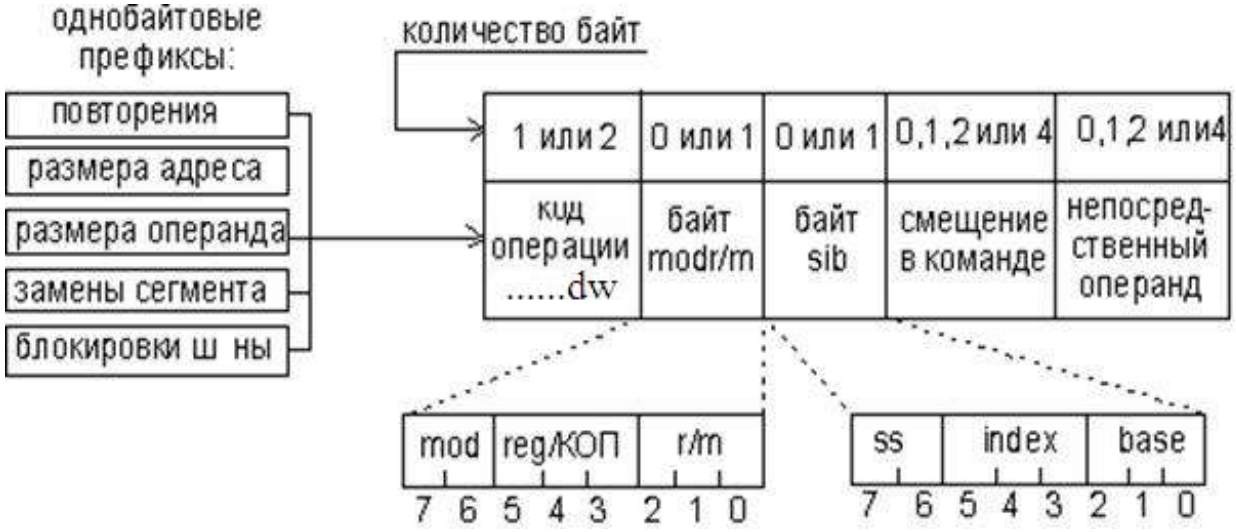

Рис. 4. Структура машинной команды IA32

На рисунке использованы следующие обозначения:

- *префикс повторения* – используется только для строковых команд;
- *префикс размера адреса* (67*h*) – применяется для изменения размера смещения: 16 бит при 32-х разрядной адресации;
- *префикс размера операнда* (66*h*) – указывается, если вместо 32-х разрядного регистра для хранения операнда используется 16-ти разрядный;
- *префикс замены сегмента* – используется при адресации данных любым сегментным регистром кроме *DS*;
- *d* – направление обработки, например, пересылки данных:
  1 – в регистр, 0 – из регистра;
- *w* – размер операнда: 1 – операнды - двойные слова, 0 – операнды - байты;
- *mod* – режим: 00 - *Disp*=0 – смещение в команде 0 байт;
  01 - *Disp*=1 – смещение в команде 1 байт;
  10 - *Disp*=2 – смещение в команде 2 байта;
  11 – операнды-регистры.
- *reg*(*r*) – обозначения регистров в зависимости от размера операнда:

w=1		*w*=0	
000	**EAX**	**000**	**AL**
001	**ECX**	**001**	**CL**
010	**EDX**	**010**	**DL**
011	**EBX**	**011**	**BL**
100	**ESP**	**100**	**AH**
101	**EBP**	**101**	**CH**
110	**ESI**	**110**	**DH**
111	**EDI**	**111**	**BH**

Если в команде используется двухбайтовый регистр (например, *AX*), то перед командой добавляется префикс изменения длины операнда (66*h*).

Различают два вида команд, обрабатывающих операнд в памяти:

- команды без байта *sib* (табл. 1);

- команды, содержащие байт *sib* (табл. 2).

Различаются эти команды по содержимому поля *m* (*r*/*m*): если *m*≠100, то байт *sib* в команде отсутствует и используется табл. 1.

Табл. 1. Схемы адресации памяти в отсутствии байта *Sib*

Поле *r*/*m*	Эффективный адрес второго операнда		
	mod = 00*b*	*mod* = 01*b*	*mod* = 10*b*
000*b*	*EAX*	*EAX+Disp8*	*EAX+Disp32*
001*b*	*ECX*	*ECX+Disp8*	*ECX+Disp32*
010*b*	*EDX*	*EDX+Disp8*	*EDX+Disp32*
011*b*	*EBX*	*EBX+Disp8*	*EBX+Disp32*
100*b*	**Определяется *Sib***	**Определяется *Sib***	**Определяется *Sib***
101*b*	*Disp32*[1]	*SS:[EBP+Disp8]*	*SS:[EBP+Disp32]*
110*b*	*ESI*	*ESI+Disp8*	*ESI+Disp32*
111*b*	*EDI*	*EDI+Disp8*	*EDI+Disp32*
[1] – особый случай – адрес операнда не зависит от содержимого регистра *EBP*, а определяется только смещением в команде (прямая адресация). *Disp* – смещение, указанное в машинной команде.			

.

Табл. 2 – Схемы адресации памяти при наличии байта *Sib*

Поле *base*	Эффективный адрес второго операнда		
	mod = 00*b*	*mod* = 01*b*	*mod* = 10*b*
000*b*	*EAX+ss*index*	*EAX+ss*index +Disp8*	*EAX+ss*index +Disp32*
001*b*	*ECX+ss*index*	*ECX+ss*index +Disp8*	*ECX+ss*index +Disp32*
010*b*	*EDX+ss*index*	*EDX+ss*index +Disp8*	*EDX+ss*index +Disp32*
011*b*	*EBX+ss*index*	*EBX+ss*index +Disp8*	*EBX+ss*index +Disp32*
100*b*	*SS:[ESP+ss*index]*	*SS:[ESP+ ss*index]+Disp8*	*SS:[ESP+ ss*index] +Disp32*
101*b*	*Disp32*[1]*+ss*index*	*SS:[EBP+ss*index +Disp8]*	*SS:[EBP+ss*index +Disp32]*
110*b*	*ESI+ss*index*	*ESI+ss*index +Disp8*	*ESI+ss*index +Disp32*
111*b*	*EDI+ss*index*	*EDI+ss*index +Disp8*	*EDI+ss*index +Disp32*
ss – масштаб; *index* – индексный регистр; *base* – базовый регистр; *Disp* – смещение, указанное в машинной команде			

Примеры:

1) `mov EBX,ECX`

100010DW Mod Reg Reg
10001001 11 001 011
8 9 C B

2) `mov BX,CX`

префикс1 100010DW Mod Reg Reg
01100110 10001001 11 001 011
6 6 8 9 C B

3) **`mov ECX,[DS:EBX+6]`**

100010DW Mod Reg Reg Смещение младший байт
10001011 01 001 011 00000110
4 B 4 B 0 6

4) **`mov CX,[DS:EBX+6]`**
префикс 100010DW Mod Reg Reg Смещение младший байт
01100110 10001011 01 001 011 00000110
6 6 8 B 4 B 0 6

5) **`mov CX,[ES:EBX+6]`**
префикс1 префикс2 100010DW Mod Reg Reg Смещение м. байт
01100110 00100110 10001011 01 001 011 00000110
6 6 2 6 8 B 4 B 0 6

6) **`mov ECX,[EBX+EDI*4+6]`**
100010DW Mod Reg Mem SS Ind Base Смещение младший байт
10001011 01 001 100 10 111 011 00000110
8 B 4 C B B 0 6

При выполнении лабораторной работы следует иметь в виду, что команды *mov*, любой из операндов которых размещен в регистрах *AL|AX|EAX*, а второй задан смещением в памяти, имеют особый формат. Это связано с тем, что указанные команды унаследованы от более старого прототипа – 16-ти разрядного процессора 8080. Эти команды не содержат байта адресации и имеют коды операции $A0_{16}$ .. $A3_{16}$, два последних бита которых также расшифровываются, как *D* и *W*. При этом *D*=1 соответствует «из регистра», а *D*=0 – «в регистр», *W* имеет те же значения, что и в предыдущем формате. Сразу за кодом операции этих команд следуют 4-х байтовые (с учетом 32-х разрядной адресации) смещения.

Примеры:

1) **`mov AL,[A]`** ; при адресе A соответствующем 0x403000

101000DW Смещение 32 разряда
10100000 00000000 00110000 01000000 00000000
A 0 0 0 3 0 4 0 0 0

2) **mov [B],AX ;** при адресе B соответствующем 0x403004

префикс1 101000DW Смещение 32 разряда
01100110 10100011 00000100 00110000 01000000 00000000
6 6 A 0 0 4 3 0 4 0 0 0

2.1.2. Команды целочисленной арифметики *IA*-32

Процессоры семейства *IA*-32 поддерживают арифметические операции (сложение, вычитание, умножение и т.п.) над однобайтовыми, двухбайтовыми и четырехбайтовыми целыми числами.

Размер операндов при этом определяется:

- объемом регистра, хранящего число, если хотя бы один операнд находится в регистре;
- машинной командой, если это указано в описании команды;
- специальными описателями, например, *byte* (байт), *word* (слово) и *dword* (двойное слово), если *ни один операнд не находится в регистре и размер операнда не определен командой*.

Перечень основных команд с описанием их форматов приведен в приложении А.

В качестве примера выполнения лабораторной работы рассмотрим программу на языке ассемблера, вычисляющую

$X = (A+B)(B-1)/(D+8)$.

Ниже приведен текст, который добавляется в соответствующие секции шаблона:

- описание инициализированных переменных в секцию *.data*:

```
A       dw    25
B       dw    -6
D       dw    11
```

- описание неинициализированных переменных в секцию *.bss*:

```
X       resw 1
```

- фрагмент кода в секцию *.text*:

```
_start:  mov    CX,[D]
         add    CX,8; CX:=D+8
         mov    BX,[B]
         dec    BX  ; BX:=B-1
         mov    AX,[A]
         add    AX,[D]; AX:=A+D
         imul   BX  ; DX:AX:=(A+D)*(B-1)
         idiv   CX  ; AX:=(DX:AX):CX
         mov    [X],AX
         . . .
```

2.1.3. Программирование ввода-вывода

Системные функции *read* и *write*, примеры обращения к которым включены в заготовки, осуществляют ввод и вывод данных в виде символьных строк. Однако из ранее изученных дисциплин вы знаете, что для выполнения арифметических операций числа должны быть представлены в памяти в одном из внутренних форматов, в которых знак, если он предусмотрен, кодируется первым битом, а само число записано в двоичной системе счисления.

Для преобразования в такое представление и обратно будем использовать специально разработанные подпрограммы, которые подробно будут рассмотрены на лекциях. Подпрограммы предназначены для лабораторных работ, а потому ограничивают значения вводимых и выводимых чисел $|x| \leq 30000$.

Подпрограмма преобразования строки, завершающейся байтом 0*x*0*A*, в число *StrToInt*.

Вход: *ESI* – адрес строки, содержащей запись числа и завершающаяся байтом 0*x*0*A* (положительные числа должны вводиться без знака, отрицательные – содержать знак в первой позиции).

Выход: *EAX* – 32-х разрядное число, *EBX* содержит 0, если преобразование прошло без ошибок, и 1, если в процессе преобразования обнаружен ввод недопустимого символа или введенное число не попадает в заданный интервал.

Подпрограмма преобразования числа в строку, завершающуюся байтом 0*x*0*A*, в число *IntToStr*.

Вход: *EAX* – число, *ESI* – адрес области памяти для размещения строки результата (7 байт).

Выход: *EAX* – размер строки результата – запись числа будет прижата к левой границе области по адресу *ESI* и завершаться байтом 0*x*0*A*.

Тексты подпрограмм преобразования приведены в приложении 2 и размещены в специальном файле *lib.asm*, который можно скачать на сайте кафедры с другими материалами по дисциплине. Этот файл необходимо поместить в каталог *labs*, созданный для хранения текстов всех лабораторных работ. Чтобы подсоединить библиотеку, находящуюся в родительском каталоге, к программе лабораторной, следует дописать в конец заготовки программы на ассемблере директиву:

```
%include "../lib.asm"
```

Для вызова подпрограмм должна использоваться машинная команда

```
call <Имя_подпрограммы>
```

Все параметры необходимо записать в соответствующие регистры.

Пример: вызов подпрограммы *StrToInt* для преобразования введенной строки в число *A*:

```
InBuf   resb    10          ; буфер ввода
lenIn   equ     $-InBuf
A       resw    1
          ...
        mov     esi,InBuf   ; адрес введенной строки
        call    StrToInt
        cmp     EBX, 0      ; проверка кода ошибки
        jne     Error       ; при преобразовании обнаружена
                            ; ошибка
        mov     [A],ax      ; запись числа в память
          ...
```

Пример: вызов подпрограммы *IntToStr* для преобразования результата в строку перед выводом:

```
OutBuf  resb    10              ; буфер вывода
X       resw    1
        …
        mov     esi, OutBuf     ; загрузка адреса буфера вывода
        mov     ax, [X]         ; загрузка числа в регистр
        cwde                    ; развертывание числа из ax в eax
        call    IntToStr
        ...
```

2.2. Порядок выполнения работы

2.2.1. Прочитайте и проанализируйте свой вариант задания. Определите последовательность вычислений и разработайте программу в соответствии с заданием своего варианта.

2.2.2. Введите программу в компьютер с использованием текстового редактора и выполните ее трансляцию и компоновку. Если при этом в программе были обнаружены ошибки, то исправьте их.

2.2.3. Покомандно просмотрите в отладчике и зафиксируйте в отчете ход выполнения вычислений. Убедитесь в правильности выполнения программы на заданных данных.

2.2.4. Посмотрите в отладчике форматы 3-4-х команд *mov* и расшифруйте двоичные коды этих команд, используя материалы лекций и раздела 2.1.1.

2.2.5. Продемонстрируйте работу программы и расшифровки команд преподавателю.

2.2.6. Составьте отчет по лабораторной работе.

2.2.7. Защитите лабораторную работу преподавателю.

2.3. Требования к отчету

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш – не допускается). Схемы и таблицы также должны быть напечатаны при помощи

компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Отчет должны иметь титульный лист, на котором указывается:

а) наименование факультета и кафедры;

б) название дисциплины;

в) номер и тема лабораторной работы;

г) фамилия преподавателя, ведущего занятия;

д) фамилия, имя и номер группы студента.

Отчет должен содержать:

1) текст задания;

2) текст программы с комментариями;

3) вручную посчитанные тесты;

4) результаты, полученные при выполнении программы;

5) выводы.

2.4 Требования к защите

Защита лабораторной предполагает ответы на вопросы преподавателя. При этом работа считается защищенной, если студент правильно и уверенно отвечает на 2-3 вопроса преподавателя, может указать в программе те или иные операции, а также поясняет работу указанного фрагмента.

Контрольные вопросы

1. Что такое машинная команда? Какие форматы имеют машинные команды процессора IA32? Чем различаются эти форматы?

2. Назовите мнемоники основных команд целочисленной арифметики. Какие форматы для них можно использовать?

3. Сформулируйте основные правила построения линейной программы вычисления заданного выражения.

4. Почему ввод-вывод на языке ассемблера не программируют с использованием соответствующих машинных команд? Какая библиотека используется для организации ввода вывода в данной лабораторной?

5. Расскажите, какие операции используют при организации ввода-вывода.

3. Лабораторная работа Программирование ветвлений и итерационных циклов

Цель работы: изучение средств и приемов программирования ветвлений и итерационных циклов на языке ассемблера.

Объем работы: 4 часа.

Пример задания. Разработать программу на языке ассемблера, вычисляющую:

$$c = \begin{cases} \frac{a}{c} - e, \text{если } a > 5 \\ c * a, \text{иначе} \end{cases}$$

Варианты заданий приведены на странице дисциплины на сайте кафедры.

3.1. Описание используемого программного обеспечения

3.1.1. Программирование ветвлений

Ветвления (рис. 5) на языке ассемблера программируют с использованием команд условной и безусловной передачи управления.

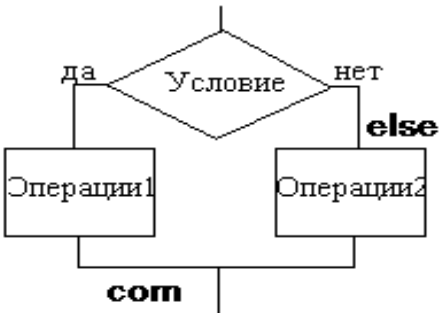

Рис. 5. Конструкция Ветвление

Так, фрагмент, показанный на рисунке, может быть реализован следующим образом:

```
        cmp  ...        ; сравнение
        j<не условие> else
        <Операции 1>
        jmp com
else:   <Операции 2>
```

```
com :     ...
```

Поскольку команды условного перехода предполагают анализ результата выполнения предыдущих команд, сначала выполняют сравнение – проверку заданного условия. В результате выполнения операции сравнения будут установлены флаги, по которым будет осуществляться переходы.

Переход, программируемый после сравнения, – условный. Запись **j**<не условие> означает условие, противоположное указанному в схеме алгоритма. Таким образом, если условие ветвления не выполняется, то осуществляется переход на фрагмент, помеченный как <Операции 2>, выполняемый в ветви «нет».

Если условие ветвления выполняется, то переход на <Операции 2> не выполняется. В этом случае после проверки условия перехода управление переходит к следующим командам фрагмента, помеченным на рисунке как <Операции 1>.

По завершению фрагмента <Операции 1> управление необходимо передать на команду, следующую за ветвлением, иначе автоматически оно будет передано командам, помеченным как <Операции 2>, что противоречит реализуемой схеме алгоритма. Дело в том, что без команды безусловной передачи управления, эти команды фактически оказываются записаны в программе после команд <Операции 1>.

Пример. Написать фрагмент вычисления *X=max(A,B)*.

```
        mov ax,[A]
        cmp ax,[B] ; сравнение A и B
        jl less ; переход по меньше
        mov [X],ax
        jmp continue ; переход на конец ветвления
less:   mov ax,[B]
        mov [X],ax
continue: ...
```

3.1.2. Программирование итерационных циклов (цикл-пока)

Программирование циклических процессов осуществляется с использованием команд переходов, или – в случае счетных циклов – с использованием команд организации циклов. Так, чтобы реализовать конструкцию Цикл-пока (рис. 6) необходим один условный и один безусловный переходы:

```
cycl:    cmp ... ; проверка условия выхода
         jne com ; выход из цикла
         <Операции> ; тело цикла
         jmp cycl ; возврат в цикл
com:  ...
```

Рис. 6. Конструкция Цикл-пока

Пример. Написать фрагмент программы суммирования чисел от 1 до 10, используя итерационный цикл.

```
         mov ax,0 ; обнуление суммы
         mov bx,1 ; первое слагаемое
cycle:   cmp bx,10 ; слагаемое больше 10
         jg continue ; выход из цикла
         add ax,bx ; суммирование
         inc bx ; следующее число
         jmp cycle ; возврат в цикл
continue: ... ; выход, сумма - в ax
```

3.2. Порядок выполнения работы

3.2.1. Прочитайте и проанализируйте свой вариант задания.

3.2.2. Разработайте схему алгоритма решения задачи.

3.2.3. Напишите соответствующую программу на языке ассемблера.

3.2.4. Введите текст программы, выполните ее трансляцию и компоновку.

3.2.5. Подберите тестовые данные (не менее 3-х вариантов) и заранее просчитайте на калькуляторе или устно ожидаемый результат.

3.2.6. Протестируйте и отладьте программу на выбранных тестовых данных.

3.2.7. Продемонстрируйте работу программы преподавателю.

3.2.8. Составьте отчет по лабораторной работе.

3.2.9. Защитите лабораторную работу преподавателю.

3.3. Требования к отчету

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш – не допускается). Схемы также должны быть напечатаны при помощи компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Каждый отчет должен иметь титульный лист, на котором указывается:

а) наименование факультета и кафедры;

б) название дисциплины;

в) номер и тема лабораторной работы;

г) фамилия преподавателя, ведущего занятия;

д) фамилия, имя и номер группы студента;

е) номер варианта задания.

Кроме того, отчет по лабораторной работе должен содержать**:**

1) схему алгоритма, нарисованную вручную или в соответствующем пакете;

2) текст программы;

3) результаты тестирования, оформленные в виде таблицы вида:

Исходные данные	Ожидаемый результат	Полученный результат

4) выводы.

3.4. Требования к защите

Защита лабораторной предполагает ответы на вопросы преподавателя. При этом работа считается защищенной, если студент правильно и уверенно отвечает на 2-3 вопроса преподавателя, может указать в программе те или иные операции, а также поясняет работу указанного фрагмента.

Контрольные вопросы

1. Какие машинные команды используют при программировании ветвлений и циклов?

2. Выделите в своей программе фрагмент, реализующий ветвление. Каково назначение каждой машинной команды фрагмента?

3. Чем вызвана необходимость использования команд безусловной передачи управления?

4. Поясните последовательность команд, выполняющих операции ввода-вывода в вашей программе. Чем вызвана сложность преобразований данных при выполнении операций ввода-вывода?

4. Лабораторная работа Обработки массивов и матриц

Цель работы: изучение приемов моделирования обработки массивов и матриц в языке ассемблера.

Объем работы: 4 часа.

Пример задания. Дана матрица 3x9. Определить среднее арифметическое отрицательных элементов каждой строки и поместить на место первого отрицательного элемента. Организовать ввод матрицы и вывод результатов.

Варианты заданий приведены на странице дисциплины на сайте кафедры.

4.1. Описание используемого программного обеспечения

4.1.1. Организация счетного цикла

Для организации счетного цикла с использованием машинной команды *LOOP* необходимо записать количество повторений в регистр счетчика *ECX*. Тогда команда *LOOP* будет отсчитывать повторения, вычитая 1 из счетчика.

Примечание. Если перед началом цикла в регистр *ECX* загружен 0, то цикл выполняется 2^{32} раз. Такая ситуация называется «зацикливанием», поскольку программа надолго «зависает».

На рис. 7 показан фрагмент схемы алгоритма, включающего счетный цикл.

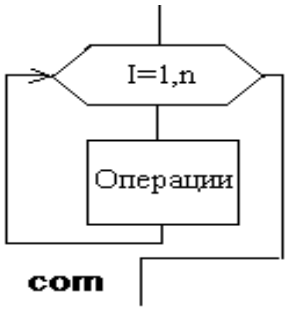

Рис. 7. Конструкция Счетный цикл

Приведенный ниже фрагмент программы на ассемблере реализует приведенную на рисунке схему алгоритма:

```
        mov ECX,[n] ; загрузка счетчика
begin_loop: <Операции> ; тело цикла
```

```
        loop begin_loop
com:    ...
```

Пример. Написать фрагмент программы суммирования чисел от 1 до 10, используя счетный цикл.

```
            mov   AX,0 ; обнуление суммы
            mov   BX,1 ; первое слагаемое
            mov   ECX,10 ; загрузка счетчика
cycle:      add   AX,BX ; суммирование
            inc   BX ; следующее число
            loop cycle ; возврат в цикл
continue:   ...          ; выход, сумма – в регистре ax
```

4.1.2. Моделирование одномерных массивов

Массив во внутреннем представлении – это последовательность элементов в памяти. В ассемблере такую последовательность можно определить, например, так:

```
A dw 10,13,28,67,0,-1 ; массив из 6 чисел размером «слово».
```

Программирование обработки выполняется с использованием адресного регистра, в котором хранится либо смещение текущего элемента относительно начала сегмента данных, либо его смещение относительно начала массива. При переходе к следующему элементу и то, и то смещение увеличивают на длину элемента. Если длина элемента отлична от единицы, то можно использовать масштаб.

Для работы с массивами часто используется специальная команда – команда загрузки исполнительного адреса:

```
LEA r32,mem
```

где *mem* – адрес в оперативной памяти,

*r*32 – 32-х разрядный регистр.

Команда вычисляет исполнительный адрес второго операнда, находящегося в памяти, и помещает его в регистр первого операнда.

Примеры:

`lea EBX,[Exword]` ; в регистр *EBX* загружается исполнительный адрес *Exword*;

`lea EBX,[EDI+10]` ; в регистр *EBX* загружается адрес 10-го байта относительно точки, на которую указывает адрес в регистре *EDI*.

Пример. Написать процедуру, выполняющую суммирование массива из 10 чисел размером слово.

Вариант 1 (используется адрес):

```
        mov AX,0
        lea EBX,[MAS]
        mov ECX,10
cycle:  add AX,[EBX]
        add EBX,2
        loop cycle
```

Вариант 2 (используется смещение):

```
        mov AX,0
        mov EBX,0
        mov ECX,10
cycle:  add AX,[EBX*2+MAS]
        add EBX,1
        loop cycle
```

Второй вариант позволяет получать более наглядный код и потому является предпочтительным, однако, к сожалению, его можно реализовать не на любом процессоре, только на процессорах ряда *x*86.

В том случае, если элементы просматриваются не подряд, адрес элемента может рассчитываться по его номеру. Так, если элементы нумерованы с единицы, то их адрес определяется как:

$A_{\text{исп}} = A_{\text{начала}}$**+(<Номер>-1)*<Длина элемента>.**

Полученный по формуле адрес записывается в 32-х разрядный регистр и используется для доступа к элементу.

Пример. Написать фрагмент, который извлекает из массива, включающего 10 чисел размером слово, число с номером *n* ($n \le 10$).

`mov EBX,[N]` ; номер числа

```
dec EBX ; вычитаем 1
mov AX,[EBX*2+MAS] ; результат в регистре AX
```

4.1.3. Моделирование матриц

Элементы матрицы могут располагаться в памяти по строкам и по столбцам. Для определенности будем считать, что матрица расположена в памяти построчно, как в языках Паскаль и *C*/*C*++.

При обработке элементов матрицы следует различать просмотр по строкам, просмотр по столбцам, просмотр по диагоналям и произвольный доступ.

Если матрица расположена в памяти по строкам и просмотр выполняется по строкам, то обработка может выполняться так, как в одномерном массиве, без учета перехода от одной строки к другой.

Пример. Написать фрагмент определения максимального элемента матрицы *A*(3,5). Размер элемента – 2 байта.

```
        mov EBX,0 ; номер элемента 0
        mov ECX,14 ; счетчик цикла
        mov AX,[A] ; заносим первое число
cycle:  cmp AX,[EBX*2+2+A] ; сравниваем числа
        jge next ; если больше, то перейти к следующему
        mov AX,[EBX*2+2+A] ; если меньше, то запомнить
next:   add EBX,1 ; переходим к следующему числу
        loop cycle
```

Просмотр по строкам при необходимости фиксировать завершение строки и просмотр по столбцам при построчном расположении в памяти выполняются в двойном цикле.

Пример. Определить сумму максимальных элементов столбцов матрицы *A*(3,5).

```
        mov AX,0 ; обнуляем сумму
        mov EBX,0 ; смещение элемента столбца в строке
```

```
        mov ECX,5 ; количество столбцов
cycle1: push ECX ; сохраняем счетчик
        mov ECX,2 ; счетчик элементов в столбце
        mov DX,[EBX+A] ; заносим первый элемент столбца
        mov ESI,10 ; смещение второго элемента столбца
cycle2: cmp DX,[EBX+ESI+A] ; сравниваем
        jge next ; если больше или равно - к следующему
        mov DX,[EBX+ESI+A] ; если меньше, то сохранили
next: add ESI,10 ; переходим к следующему элементу
        loop cycle2 ; цикл по элементам столбца
        add AX,DX ; просуммировали максимальный элемент
        pop ECX ; восстановили счетчик
        add EBX,2 ; перешли к следующему столбцу
        loop cycle1 ; цикл по столбцам
```

При просмотре по диагонали обычно строится один цикл. При этом для адресации элементов используется специальный регистр смещения, который должен соответствующим образом переадресовываться.

4.2. Порядок выполнения работы

4.2.1. Прочитайте и проанализируйте свой вариант задания.

4.2.2. Разработайте схему алгоритма решения задачи.

4.2.3. Напишите программу.

4.2.4. Введите текст программы в компьютер, выполните ее трансляцию и компоновку.

4.2.5. Подберите тестовые данные (не менее 3-х вариантов).

4.2.6. Протестируйте и отладьте программу на выбранных тестовых данных.

4.2.7. Продемонстрируйте работу программы преподавателю.

4.2.8. Составьте отчет по лабораторной работе.

4.2.9. Защитите лабораторную работу преподавателю.

4.3. Требования к отчету

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш – не допускается). Схемы также должны быть напечатаны при помощи компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Каждый отчет должен иметь титульный лист, на котором указывается:

а) наименование факультета и кафедры;

б) название дисциплины;

в) номер и тема лабораторной работы;

г) фамилия преподавателя, ведущего занятия;

д) фамилия, имя и номер группы студента;

е) номер варианта задания.

Кроме того отчет по лабораторной работе должен содержать**:**

1) схему алгоритма, выполненную вручную или в соответствующем пакете;

2) текст программы;

3) результаты тестирования, которые должны быть оформлены в виде таблицы вида:

Исходные данные	Ожидаемый результат	Полученный результат

4) выводы.

4.4. Требования к защите

Защита лабораторной предполагает ответы на вопросы преподавателя. При этом работа считается защищенной, если студент правильно и уверенно отвечает на 2-3 вопроса преподавателя, может указать в программе те или иные операции, а также поясняет работу указанного фрагмента.

Контрольные вопросы

1. Почему в ассемблере не определены понятия «массив», «матрица»?

2. Как в ассемблере моделируются массивы?

3. Поясните фрагмент последовательной адресации элементов массива? Почему при этом для хранения частей адреса используют регистры?

4. Как в памяти компьютера размещаются элементы матриц?

5. Чем моделирование матриц отличается от моделирования массивов? В каких случаях при выполнении операций для адресации матриц используется один регистр, а в каких – два?

5. Лабораторная работа Связь разноязыковых модулей

Цель работы: изучение конвенций о способах передачи управления и данных при вызове из программы, написанной на языке высокого уровня, подпрограмм, написанных на ассемблере.

Объем работы: 5 часов.

Пример задания. Дан текст не более 255 символов. Слова отделяются друг от друга пробелами. Расставить слова по возрастанию сумм кодов входящих в них символов.

Разработать программу, состоящую из трех модулей (два на *Free Pascal* или *C++* и один - на ассемблере):

♦ основной модуль на заданном языке программирования общего назначения должен содержать диалоговый ввод необходимых данных, вызов функции или процедуры на ассемблере и вывод результатов;

♦ второй модуль - функция или процедура на ассемблере, выполняющая заданную обработку и вызывающая для печати диагностических сообщений процедуру на выбранном языке программирования общего назначения;

♦ третий модуль – процедура на заданном языке общего назначения, обязательно получающая некоторые параметры из вызывающего модуля.

Язык основной программы и третьего модули уточняется в задании. Варианты заданий приведены на странице дисциплины на сайте кафедры.

5.1. Описание используемого программного обеспечения

5.1.1. Проблемы связи разноязыковых модулей

Основные проблемы связи разноязыковых модулей:

- осуществление совместной компоновки модулей;
- организация передачи и возврата управления;
- передача параметров:
  - с использованием глобальных переменных,
  - с использованием стека и регистров;

- обеспечение возврата результата функции;
- обеспечение правильного использования регистров процессора.

При компоновке модулей следует учитывать особенности компиляторов различных языков программирования и их реализаций в конкретных средах.

Согласование типа вызова в настоящей работе не выполняется, поскольку во всех случаях используется модель памяти *Flat*, для которой все вызовы ближние *near*, но смещение имеет размер 32 бита или 64 бита в зависимости от используемого оборудования и разрядности операционной системы.

Передача параметров и возврат результатов декларируются специальными конвенциями, называемыми «конвенциями о связи». Эти правила определяют способ передачи параметров, закономерности формирования внутренних имен подпрограмм и глобальных данных и модель памяти.

Главное правило использования регистров устанавливает, что все регистры, кроме специально оговоренных, должны на выходе из подпрограммы содержать те же значение, что и при входе в нее. Поэтому значения регистров, изменяемые в подпрограммах, при входе в подпрограмму должны сохраняться в стеке, а при выходе – восстанавливаться из него.

5.1.2. Конвенции о связи

Конвенции о связи декларируют способы передачи параметров при организации связи вызывающей и вызываемой подпрограмм (модулей). При программировании для операционной системы *Windows* используют пять конвенций: паскаль, си, стандартная *Windows*, защищенная и регистровая (табл. 3).

Табл. 3. Конвенции *Windows* по передаче параметров

	Конвенция	*Delphi*	*C++ Builder* и *Visual C++*	Порядок параметров в стеке	Очистка стека	Использование регистров
1	Паскаль	*pascal*	*_ _pascal*	Слева направо	Вызываемая процедура	Нет
2	Си	*cdecl*	*_ _cdecl*	Справа налево	Вызывающая программа	Нет
3	Стандартная	*stdcall*	*_ _ stdcall*	Справа налево	Вызываемая процедура	Нет
4	Защищенная	*savecall*	–	Справа налево	Вызываемая процедура	Нет
5	Регистровая	*register*	*_ _fastcall*	Справа налево	Вызываемая процедура	Три регистра *EAX*, *EDX*, *ECX* (*VS* – до 2-х), остальные параметры – в стеке

В отличие от *Windows* операционные системы *Linux Astra*, как и другие *Unix*-системы, использует единственную конвенцию, которая включена в стандартизованный двоичный интерфейс приложения (англ. *System V Application Binary Interface* или сокращенно *System V ABI*). Кроме соглашений о вызовах, интерфейс диктует форматы объектных и исполняемых файлов (формат исполняемых и связываемых файлов *ELF* также является частью *System V ABI*) и многое другое для систем, следующих спецификации *X*/*Open Common Application Environment Specification* и определению интерфейса *System V*.

Указанная конвенция во многом аналогична регистровой, но предполагает, что до 6-ти параметров передается в регистрах, остальные – в стеке в обратном порядке. Порядок занесения параметров в регистры прямой, т.е. первым записывается в регистр первый параметр и т.д.

В 32-х разрядной программе для передачи параметров используют регистры *EBX*, *ECX*, *EDX*, *ESI*, *EDI*, *EBP*; функции, за исключением функций, результатом которых является строка, возвращают результат в регистре *EAX*. Если функция возвращает строку, то адрес строки-результата передается в регистре *EBX*, т.е. в первом регистре, используемом для передачи параметров.

В 64-х разрядной программе для передачи параметров используют регистры *RDI*, *RSI*, *RDX*, *R10*, *R8*, *R9*; функции, за исключением функций, результатом которых является строка, возвращают результат в регистре *RAX*. Если функция возвращает строку, то адрес строки-результата передается в регистре *RDI*, т.е. в первом регистре, используемом для передачи параметров.

Вызов 32-х разрядной подпрограммы реализуется по варианту:

```
;  занесение параметров в регистры
      mov   ebx, <Параметр 1>
      mov   ecx, <Параметр 2>
       …
      call  <Имя подпрограммы>   ;   вызов подпрограммы
```

Вызов 64-х разрядной подпрограммы аналогичен с точностью до используемых для передачи параметров регистров:

```
;  занесение параметров в регистры
      mov   rdi, <Параметр 1>
      mov   rsi, <Параметр 2>
       …
      call  <Имя подпрограммы>   ;   вызов подпрограммы
```

Вызываемые 32-х или 64-х разрядные подпрограммы должны иметь стандартно оформленные вход – *пролог* и выход – *эпилог*.

Пролог 64-х разрядной подпрограммы:

```
<Имя подпрограммы>:
      push  RBP    ;сохранить старое RBP в стеке
      mov   RBP,RSP  ;установить базу для параметров в стеке
```

```
lea     RSP,[RSP-<Объем памяти локальных переменных>]
<Копирование параметров в стек>
<Сохранение используемых регистров>
```

…

Эпилог:

```
…
<Восстановление используемых регистров>
mov     RSP,RBP ; удалить область локальных переменных
pop     RBP     ; восстановить значение RBP
ret
```

Пролог 32-х разрядной программы идентичен приведенному с точностью до имен 32-х или 64-х разрядных регистров.

Таким образом в момент получения управления 64-х разрядной подпрограммой в регистрах находятся до 6-ти параметров в виде значений или адресов, а в стеке – 64-х разрядный адрес возврата в вызывающую программу (рис. 8, *а*). Затем вызываемая подпрограмма, выполняя пролог, размещает в стеке значение *RBP* вызывающей программы копии параметров и область локальных переменных подпрограммы. Далее подпрограмма использует стек для своих надобностей.

При этом адресация копий параметров и локальных переменных выполняется относительно базы в регистре *RBP*. С учетом того, что стек растет в сторону младших адресов, адреса копий параметров и локальных переменных меньше адреса в регистре *RBP*. Для текущей работы со стеком по умолчанию используется регистр *RSP*, всегда содержащий адрес последнего помещенного в стек значения (рабочая область стека подпрограммы на рис. 8, *б*).

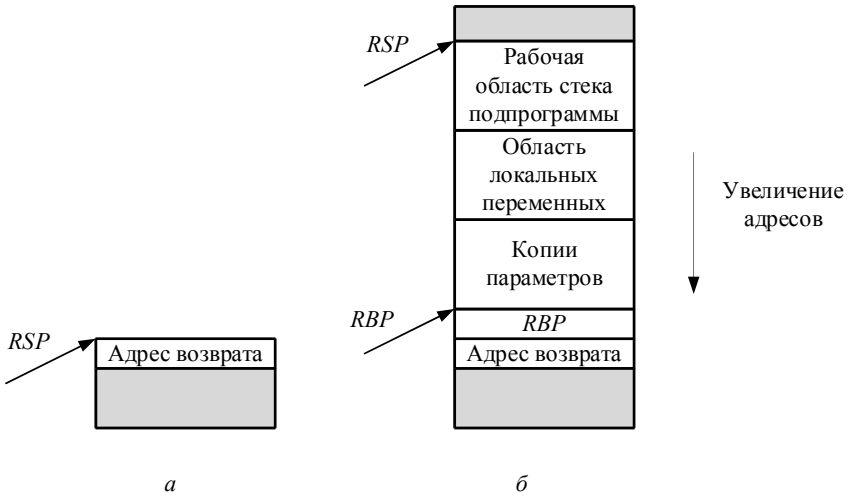

Рис. 8. Содержимое стека 64-х разрядной программы:

а – в момент передачи управления подпрограмме; *б* – во время работы подпрограммы

При выполнении эпилога содержимое из стека извлекается в обратном порядке. Если в вызываемой подпрограмме допущена ошибка и в момент выполнения команды *ret* на вершине стека находится не адрес возврата, то в момент возврата будет фиксироваться ошибка (чаще всего ошибка адресации), обычно называемая «разрушением» стека.

Особенности настройки сред и работы с ассемблером в разных средах рассмотрены в следующих пунктах.

5.1.3. Вызов ассемблерной подпрограммы из программы на *Free Pascal* (среда *Lasarus*)

5.1.3.1. Соответствие представлений данных

Язык *Pascal* использует следующие внутренние представления данных.

Целое –

`shortint`: -128..127 – байт со знаком;

`byte`: 0..255 – байт без знака;

`smallint`: -32768..32767 – слово со знаком;

`word`: 0..65535 – слово без знака;

`integer, longint`: – двойное слово со знаком.

Символ – **char** – код, байт без знака.

Булевский тип – **boolean** – 0(*false*) и 1(*true*) – байт без знака.

Указатель – **pointer** – 32-х или 64-х разрядное смещение.

Строка – **shortstring** – символьный вектор указанной при определении длины, содержащий текущую длину в первом байте.

Массив – **array** – последовательность элементов указанного типа, расположенных в памяти таким образом, что правый индекс возрастает быстрее левого (для матрицы – построчно).

Для обращения к данным этих типов в программе на ассемблере необходимо использовать определенные типы переменных (табл. 4).

Табл. 4. Соответствие типов ассемблера *Nasm* и языка *Free Pascal*

№	Описание данных в ассемблере *Nasm*	Размер памяти	Соответствующий тип данных *Free Pascal*
1	**db, resb**	1 байт	**Shortint, byte, char, boolean**
2	**dw, resw**	2 байта	**SmallInt, word**
3	**dd, resd**	4 байта	**Single, указатель, integer**
4	**dq, resq**	8 байт	**Double, comp**
5	**dt, rest**	10 байт	**Extended**

В языке *Pascal* параметры могут передаваться двумя способами: по значению и по ссылке (с указанием *var* или *const*). В первом случае подпрограмме передаются копии значений параметров, и, соответственно, она не имеет возможности менять значения передаваемых параметров в вызывающей программе. Во втором случае подпрограмма получает адреса передаваемых значений и может не только читать значения, но и возвращать в вызывающую программу измененные значения. И в том, и в другом случае параметры или их адреса заносятся в регистры, как сказано в пункте 5.1.2.

5.1.3.2. Создание и ассемблирование программного модуля на ассемблере

Текст подпрограммы на ассемблере можно набирать сразу в редакторе среды *Lazarus*. При этом созданный файл следует сохранить с суффиксом *.asm* в каталоге проекта.

Выполнять трансляцию подпрограммы на ассемблере можно в эмуляторе терминала, но гораздо эффективнее настроить в среде *Lazarus* внешний инструмент. Для этого необходимо выбрать пункт меню **Сервис/Настроить внешние средства**. В окне настройки вводим:

Заголовок: **`NASM`**

Имя файла программы: **`/usr/bin/nasm`**

Параметры: **`-f elf64 $EdFile()`**

`-l $Path($EdFile())$NameOnly($EdFile()).lst`

Рабочий каталог: **`$(ProjPath)`**

Также выбираем галочкой пункт «Искать сообщения *FPC* в выводе».

Теперь для трансляции ассемблерной подпрограммы достаточно перейти на ее вкладку в редакторе и выбрать пункт меню Сервис/*NASM*. Если трансляция пройдет без ошибок, то объектный модуль и листинг будут созданы в том же каталоге, где находился исходный модуль на ассемблере. При наличии ошибок будет создан только листинг.

Внимание! Следует помнить, что транслируется файл, который хранится на диске, а не текст, который вы видите перед собой в окне редактора, т.е. все внесенные изменения перед трансляцией ассемблерного модуля необходимо сохранить на диске!

5.1.3.3. Компиляция программы на Free Pascal и совместная компоновка с подпрограммой на ассемблере

При компиляции программы на *Free Pascal* необходимо отключить оптимизацию кода. В противном случае обычно небольшая подпрограмма на *Free Pascal*, которая должна вызываться из ассемблера, скорее всего, будет объяв-

лена компилятором *FPC* подставляемой (*inline*). Это приведет к включению операторов тела подпрограммы в место вызова без создания подпрограммы.

Для отключения оптимизации следует вызвать пункт меню Проект/Параметры проекта/Компиляция и компоновка. В открывшемся окне необходимо выбрать уровень оптимизации 0 (без оптимизации).

В соответствии с заданием в лабораторной необходимо разработать вызываемую из основной программы на *Free Pascal* ассемблерную подпрограмму, которая в свою очередь вызывает подпрограмму на *Free Pascal*.

Для выполнения совместной компоновки подпрограмм такой программы следует:

- объявить подпрограмму на *Free Pascal*, которая будет вызываться из ассемблерной подпрограммы, вызываемой извне модуля, для этого в секцию *interface* следует включить ее прототип, например

```
interface
   procedure ggg(n:integer);
```

- объявить подпрограмму на ассемблере внешней и указать источник, содержащий подпрограмму, для этого в секцию *implementation* приложения необходимо добавить прототип ассемблерной подпрограммы с описателем *external* и директиву подгрузки объектного модуля, например

```
implementation
{$l test.o}
procedure Str1(S:shortstring;var Sr:shortstring); external;
```

Следует также иметь в виду, что компилятор *FPC* при обработке исходного модуля меняет внутренние имена подпрограмм и данных. Так глобальным подпрограммам, описываемым в интерфейсной части модулей, присваиваются имена, в которых:

- все буквы имен преобразованы в прописные;

- перед именем подпрограммы указано имя модуля, содержащего ее определение, а после – список типов используемых параметров.

При этом меняется или не меняется имя ассемблерной подпрограммы, объявленной внешней в среде *Lazarus*, зависит от конкретного программного обеспечения, но имя подпрограммы на *Free Pascal*, вызываемой из ассемблера, меняется всегда.

Точные имена подпрограмм можно посмотреть после компиляции модуля, к которому подключается подпрограмма на ассемблере. Для этого надо загрузить код объектного модуля в шестнадцатеричный текстовый редактор *GHex* (меню Пуск пункт Разработка/*GHex*) и поискать в нем имена интересующих вас подпрограмм на *Free Pascal*: подпрограммы на ассемблере и подпрограммы на *Free Pascal*, которую вы собираетесь вызывать из ассемблера. Первое вхождение имени может не содержать префикса и суффикса, зато второе позволяет выделить полное имя подпрограммы (рис. 9).

Пример. Разработать подпрограмму *Str*1на ассемблере, которая выбирает из исходной строки 1-й, 3-й и 5-й символы, заносит их в результирующую строку и возвращает результат по адресу второго параметра. Также подпрограмма должна вызывать процедуру *GGG* на *Free Pascal* для вывода числа 3.

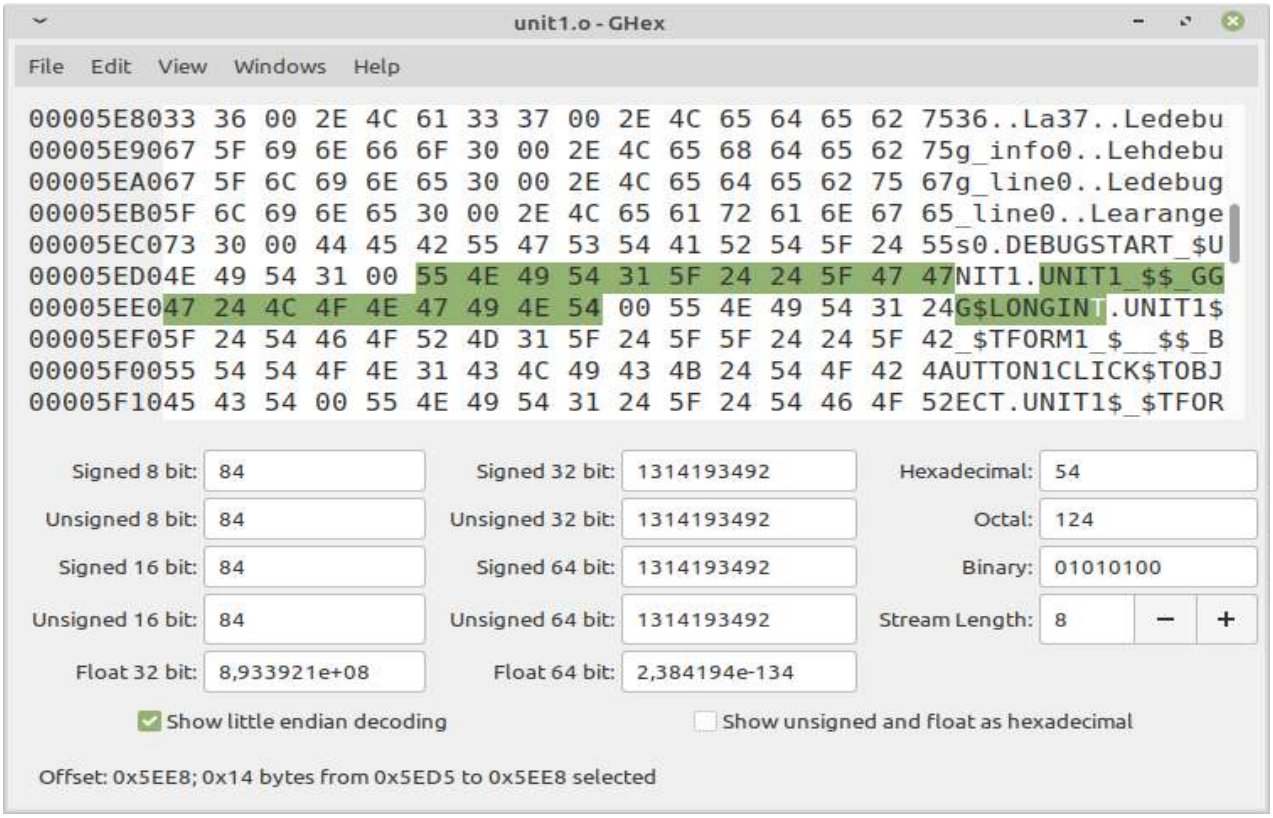

Рис. 9. Поиск имени подпрограммы *GGG* в файле *unit1.o* с использованием шестнадцатеричного редактора *ghex* (слева выделен шестнадцатеричный код имени, справа — его символьное представление)

Текст подпрограммы:

```
global Str1
extern UNIT1_$$_GGG$LONGINT
section .text
Str1:  push rbp
    mov rbp,rsp
    push rcx
    push rax
    mov byte[rsi],3
    mov ecx,3
    inc  rdi
    inc  rsi
.cycle mov al,[rdi]
    mov[rsi],al
```

```
inc rdi
inc rdi
inc rsi
loop .cycle
mov rdi,3
call UNIT1_$$_GGG$LONGINT
pop rax
pop rcx
mov rsp,rbp
pop rbp
ret
```

5.1.3.4. Отладка ассемблерных подпрограмм

Для выполнения отладки ассемблерной подпрограммы перед выполнением приложения необходимо установить точку останова на строке вызова этой подпрограммы. Когда в процессе выполнения компьютер дойдет до этой точки, он остановится. В этот момент следует открыть отладочное окно ассемблера, используя пункт меню Вид/Окна отладки/Ассемблер, и окно регистров, используя Вид/Окна отладки/Регистры.

Теперь, с помощью *локального меню* (!!!) окна Ассемблер, можно по шагам выполнять ассемблерную подпрограмму, заходя или не заходя в другие подпрограммы и отслеживая содержимое регистров.

В среде *Lazarus* для отладки программ использован отладчик *GDB*. По умолчанию этот отладчик в окне Ассемблер показывает дисассемблированную программу в синтаксисе *A&T*. Чтобы он показывал программу в синтаксисе *Intel*, следует войти в настройки Сервис/Параметры…/ Отладчик/Общие найти в этом окне параметр *Assembler Style* и выбрать для него значение *gdasIntel*.

К сожалению, в отладчике не предусмотрена возможность удобного просмотра содержимого области данных в режиме отладки на уровне машинных

команд. Однако имеется возможность через меню панели Отладчика перейти по адресу, введенному в специальном поле. Адрес в это поле копируется из окна Регистры. После перехода по адресу отладчик попытается дисассемблировать код, но при этом в шестнадцатеричном виде демонстрирует, что записано в сегменте данных в этом месте.

5.1.4. Вызов ассемблерной подпрограммы из программы на языке C++ (компиляторы *CLang* или *g++*, среда *Qt Creator*)

5.1.4.1 Соответствие представлений данных

Язык программирования *C++* использует следующие внутренние представления данных.

Целое –

`signed char`: -128..127 – байт со знаком;

`unsigned char`: 0..255 – байт без знака;

`signed short`: -32768..32767 – слово со знаком;

`unsigned short`: 0..65535 – слово без знака;

`int, long`: – двойное слово со знаком.

Символ – **`char`** – код, байт без знака.

Булевский тип – **`bool`** – 0 (*false*) и 1 (*true*) – байт без знака.

Указатель – 32-х или 64-х разрядное смещение в зависимости от разрядности кода программы.

Строка – **`char <имя> [<длина>]`**– символьный вектор указанной при определении длины, заполненная часть завершается нулем.

Массив – последовательность элементов указанного типа, расположенных в памяти таким образом, что правый индекс возрастает быстрее левого (для матрицы - построчно).

Для обращения к данным этих типов в программе на ассемблере необходимо использовать соответствующие типы переменных. Соответствие типов представлено в табл. 5.

Табл. 5. Соответствие типов ассемблера и языка *C++*

№	Типы данных ассемблера	Размер памяти	Соответствующий тип данных *Clang* или *g++*
1	**db, resb**	1 байт	**signed/unsigned char**
2	**dw, resw**	2 байта	**signed/unsigned short**
3	**dd, resd**	4 байта	**int, long**

5.1.4.2. Создание и ассемблирование подпрограммного модуля на ассемблере

Аналогично тому, как это может быть сделано в среде *Lazarus*, в среде *Qt Creator* можно создать внешний инструмент для подключения транслятора ассемблера.

Для создания внешнего инструмента следует выбрать пункт меню Инструменты/Внешние/Настроить… **.** В результате появляется окно настройки внешних утилит. Внизу окна есть кнопка Добавить. При нажатии на эту кнопку появляется подменю: добавить категорию или добавить утилиту. Вначале добавляем новую категорию. Имя категории корректируется при двойном щелчке по ее временному имени после добавления. В этом поле вводится имя Ассемблер. Аналогично добавляется утилита, которой присваивается имя *NASM*. В описании утилиты следует ввести:

Описание: **NASM**.

Программа: **/usr/bin/nasm**

Параметры: **-f elf64 %{CurrentDocument:FilePath}**

-l %{CurrentDocument:Path}/%{CurrentDocument:

FileBaseName}.lst

Рабочий каталог: (оставляем пусто)

Стандартный вывод: **Показать в консоли**

Вывод ошибок: **Показать в консоли**

Внимание! При копировании параметров в окно настройки инструмента необходимо убрать лишние пробелы перед *FileBaseName.lst*.

После этого можно создать ассемблерный модуль в редакторе среды, а также корректировать и транслировать его, не выходя из среды.

Внимание! Следует помнить, что транслируется не то, что представлено в окне, а файл на диске, поэтому каждый раз перед трансляцией модуля необходимо сохранять содержимого окна в файл.

5.1.4.3. Построение файла проекта

Совместную компоновку подпрограммы на ассемблере с программой и подпрограммой на *C++* в среде *Qt Creator* и другие настройки среды для выполнения лабораторной работы организуют с помощью файла проекта (*.pro*).

Команды, указанные в файле проекта, в нашем случае должны обеспечить:

- включение исходного файла на ассемблере в проект, чтобы его можно было вызывать в текстовый редактор через окно Навигатора, как и остальные исходные файлы программы;

- совместную компоновку программы на языке *C++* и объектного модуля, сгенерированного ассемблером;

- отключение оптимизации кода программы.

Необходимость отключения оптимизации связана с тем, что вызываемая из ассемблера, написанная в соответствии с заданием простенькая подпрограмма на *C++* в результате оптимизации скорее всего будет объявлена компилятором встраиваемой (*inline*), следовательно, операторы тела подпрограммы будет просто вставляться в текст без оформления подпрограммы, а потому компоновщик этой подпрограммы просто не найдет.

Пример файла проекта *.pro* для программы лабораторной работы в виде консольного приложения:

```
CONFIG += c++11 console   # консольное приложение
CONFIG -= app_bundle
DEFINES += QT_DEPRECATED_WARNINGS
SOURCES += main.cpp
OBJECTS += text.o # подключение объектного модуля ассемблерной
                  # подпрограммы
DISTFILES += text.asm # включение в проект исходного модуля
```

```
                                # на ассемблере для удобства вызова его
                                # исходного текста в текстовый редактор
CONFIG ~= s/-O[0123s]//g # отключение оптимизации
CONFIG += -O0
```

5.1.4.4. Формирование внутренних имен подпрограмм

Компиляторы *CLang* или *g++* языка *C++,* используемые в среде *Qt Creator*, изменяют имена всех глобальных («*extern*») переменных программы, добавляя перед ними символ подчеркивания «_», и дописывает к именам функций специальные комбинации символов, отражающие типы передаваемых параметров.

Для того, чтобы определить имя подпрограммы на ассемблере, по которому ее будет вызывать основная программа на языке *C++*, надо открыть объектный файл, созданный *Qt Creator*, используя шестнадцатеричный текстовый редактор *GHex* (меню Пуск, раздел Разработка) и найти там имя подпрограммы. Например, для процедуры

```
void sum(int x,int y, int *p);
```

в файле *main.o* (рис. 10) находим имя подпрограммы *_Z3sumiiPi.* Именно это имя и надо присвоить подпрограмме на ассемблере.

Анализ некоторого количества примеров показал, что измененное имя строится по следующему правилу:

_*Z*<Целое число><Имя подпрограммы><Описание параметров>,

где

<Целое число> – вероятно порядковый номер подпрограммы;

<Описание параметров> — строка кодов:

i — параметр целого типа;

c — параметр символьного типа;

Pi — указатель на параметр целого типа;

Pc — указатель на символ;

PKc — указатель на константный символ.

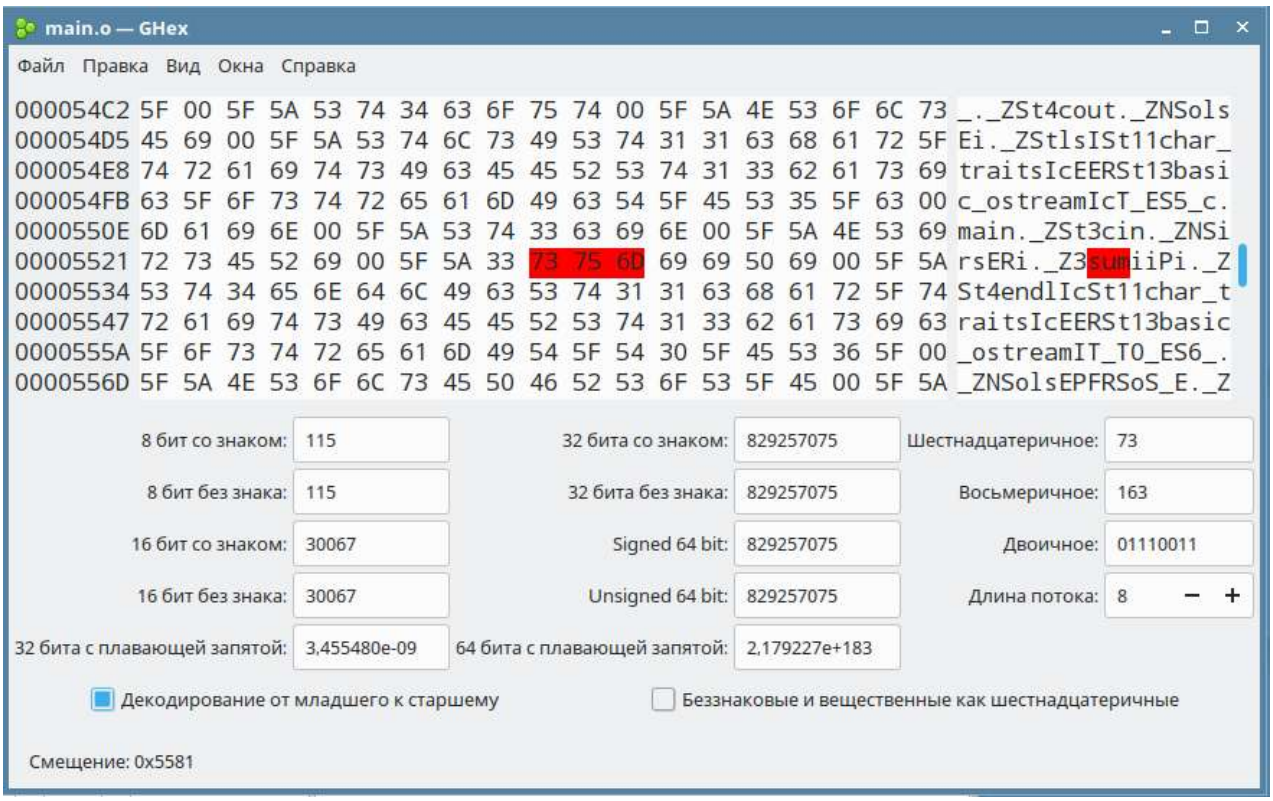

Рис. 10. Поиск измененного имени подпрограммы в объектном файле, созданном *Qt Creator*

При написании исходного модуля на ассемблере следует:

- назначить корректные имена ассемблерной подпрограммы и вызываемой из нее подпрограммы на *C++*;

- описать вызываемую из ассемблера подпрограмму на *C++*, как внешнюю *extern* **.**

Пример. Разработать подпрограмму на ассемблере *str*1, которая выбирает из исходной строки 1-й, 3-й и 5-й символы, заносит их в результирующую строку и возвращает результат по адресу второго параметра. Также подпрограмма должна вызывать процедуру *print*1 на языке *C++* для вывода целого числа на экран:

```
void print1(int n)
{ std::cout<<n<<'\n';}
```

Текст подпрограммы:

```
global _Z4str1PKcPc
extern _Z6print1i
```

```
  section .text
_Z4str1PKcPc:  push rbp
    mov rbp,rsp
    push rcx
    push rax
    mov byte[rsi],3
    mov ecx,3
.cycle mov al,[rdi]
    mov[rsi],al
    inc rdi
    inc rdi
    inc rsi
    loop .cycle
    mov byte[rsi],0
    mov rdi,3
    call _Z6print1i
    pop rax
    pop rcx
    mov rsp,rbp
    pop rbp
    ret
```

5.1.4.5. Отладка программ с ассемблерной подпрограммой

При отладке программ, содержащих ассемблерные подпрограммы, используют окно дисассемблера, окно, в котором высвечивается содержимое регистров, и окно, в котором выводится содержимое области данных.

Для того чтобы раскрыть эти окна в среде *Qt Creator*, необходимо:

- поставить точку останова на вызове ассемблерной подпрограммы;

- запустить программу в режиме отладки;

- когда программа остановится на точке останова:

--нажать на кнопку переключения в режим отладки на уровне инструкций, расположенную в заголовке панели Отладчик (рис. 11),

-- нажать на кнопку Обзоры в конце заголовка той же панели и поставить галочку на выборе панели Регистры.

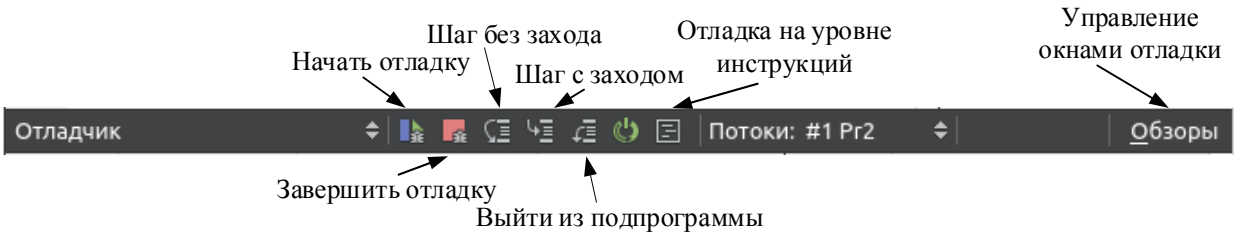

Рис. 11. Заголовок панели отладчика с локальным меню

При этом в окне среды появятся панели *Disassembler* и Регистры (рис. 12).

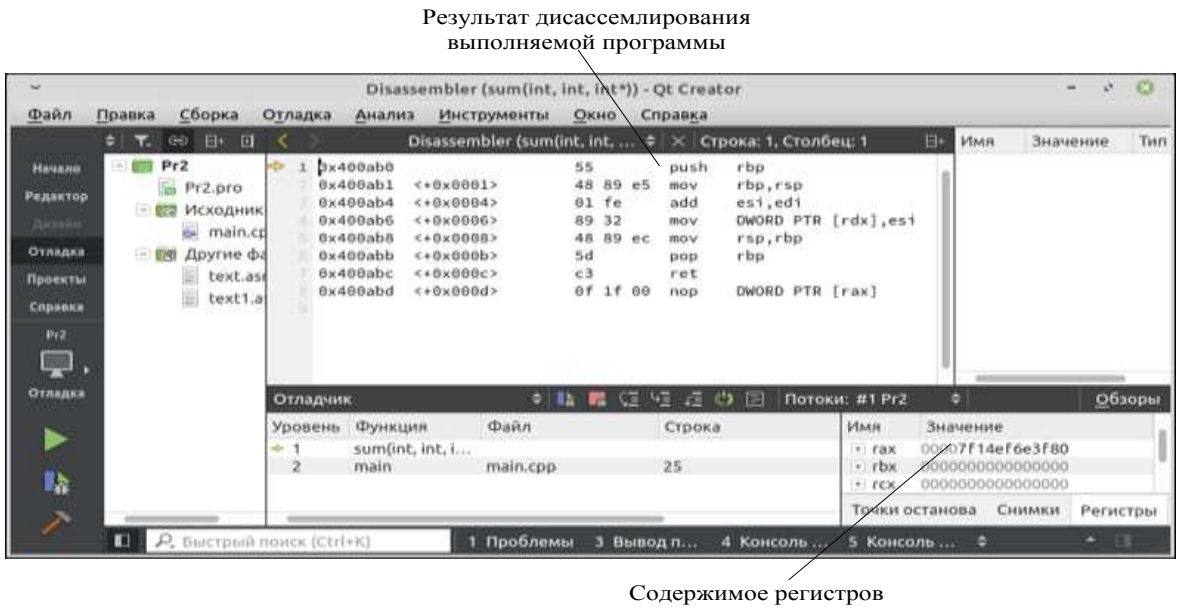

Рис. 12. Вид окна *Qt Creator* в режиме отладки на уровне инструкций

Для пошагового выполнения программы в режиме *Disassembler* следует использовать кнопки Шаг без захода, Шаг с заходом и др., расположенные в заголовке панели Отладчик (см. рис. 11).

Для просмотра содержимого памяти по адресу, записанному в одном из регистров, следует щелкнуть правой кнопкой мыши по содержимому регистра и в появившемся контекстном меню выбрать пункт Открыть просмотровщик памяти, начиная с со значения регистра… После этого на экране появится окно шестнадцатеричного дампа памяти (рис. 13).

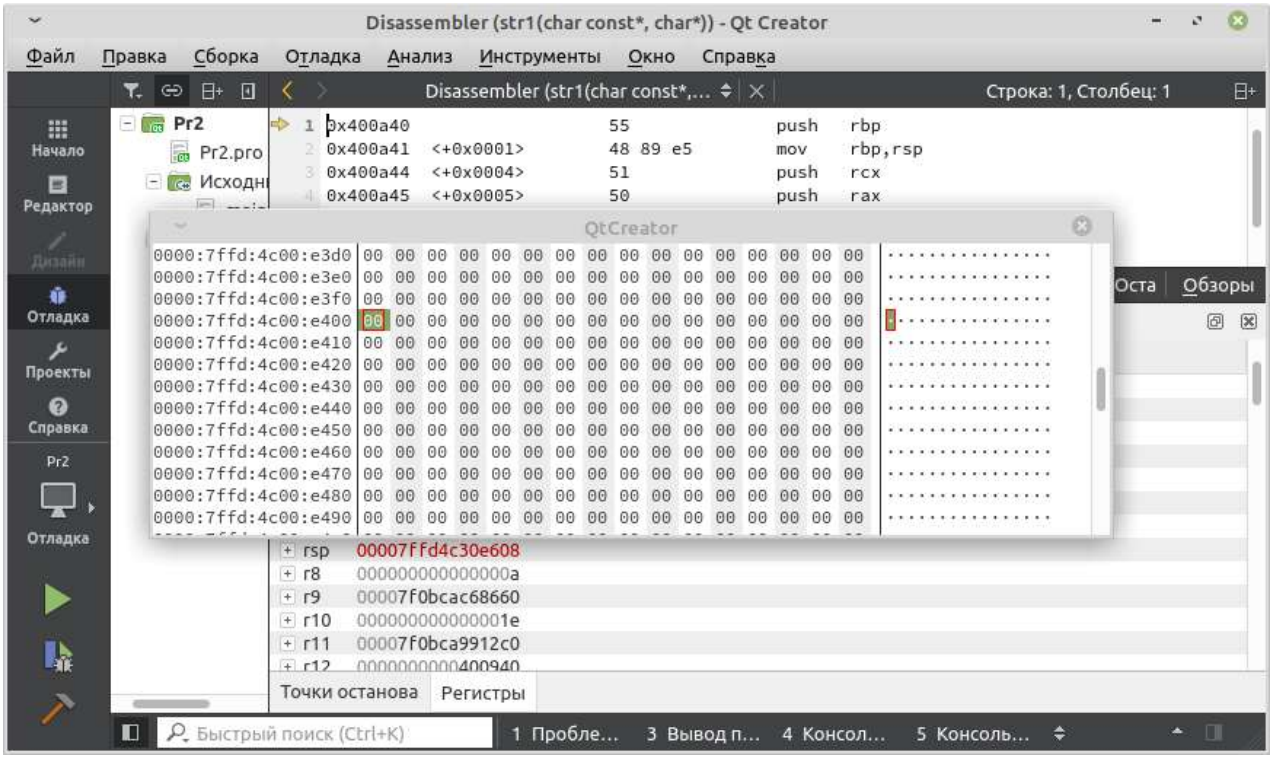

Рис. 13. Окно просмотра данных по указанному адресу (красным показано содержимое регистра, в красной рамке — первый байт по указанному адресу)

5.2. Порядок выполнения работы

5.2.1. Прочитайте и проанализируйте свой вариант задания.

5.2.2. Выполните декомпозицию программы в соответствии с заданием.

5.2.3. Разработайте схемы алгоритмов всех модулей программы.

5.2.4. Выберите среду реализации программы лабораторной работы и выполните необходимые настройки среды.

5.2.5. Составьте основную программу и подпрограмму, вызываемую из подпрограммы на языке ассемблера, на соответствующем языке высокого уровня. Введите тексты программы и подпрограммы в компьютер, используя редактор среды программирования.

5.2.6. Выполните компиляцию указанных частей программы и определите имена, которые следует присвоить подпрограмме на ассемблере и вызываемой из нее подпрограмме на языке высокого уровня.

5.2.7. Напишите программу на языке ассемблера, введите ее в компьютер, используя редактор среды программирования, и выполните ее трансляцию.

5.2.8. Выполните совместную компоновку частей программы.

5.2.9. Выполните тестирование и отладку программы на выбранных тестовых данных.

5.2.10. Продемонстрируйте работу программы преподавателю.

5.2.11. Составьте отчет по лабораторной работе.

5.2.12. Защитите лабораторную работу преподавателю.

5.3. Требования к отчету

Все записи в отчете должны быть либо напечатаны на принтере, либо разборчиво выполнены от руки синей или черной ручкой (карандаш – не допускается). Схемы также должны быть напечатаны при помощи компьютера или нарисованы с использованием чертежных инструментов, в том числе карандаша.

Каждый отчет должен иметь титульный лист, на котором указывается:

а) наименование факультета и кафедры;

б) название дисциплины;

в) номер и тема лабораторной работы;

г) фамилия преподавателя, ведущего занятия;

д) фамилия, имя и номер группы студента;

е) номер варианта задания.

Кроме того, отчет по лабораторной работе должен содержать**:**

1) схемы алгоритма всех частей программы, выполненные вручную или в соответствующем пакете;
2) схематическое изображение содержимого стека в момент передачи управления;
3) текст подпрограммы на ассемблере и фрагменты текста программы на языке высокого уровня, которые отвечают за передачу управления подпрограмме;
4) результаты тестирования, которые должны быть оформлены в виде таблицы вида:

Исходные данные	Ожидаемый результат	Полученный результат

5) выводы.

5.4. Требования к защите

Защита лабораторной предполагает ответы на вопросы преподавателя. При этом работа считается защищенной, если студент правильно и уверенно отвечает на 2-3 вопроса преподавателя, может указать в программе те или иные операции, а также поясняет работу указанного фрагмента.

Контрольные вопросы

1. Что такое «конвенции о связи»? В чем заключается конвенция *register*?

2. Что такое «пролог» и «эпилог»? Где располагается область локальных данных?

3. Как связана структура данных стека в момент передачи управления и текст программы и подпрограмм?

4. С какой целью применяют разноязыкорвые модули в одном проекте?

Литература

Иванова Г.С. Презентации курса лекций Машинно-зависимые языки и основы компиляции. М.: МГТУ им. Н.Э. Баумана, 2022. Способ доступа: http://e-learning.bmstu.ru/moodle/mod/resource/view.php?id=35 .

Канеут М.В. Free Pascal: Руководство программиста. Глава 6. Способ доступа: http://freepascal.ru/download/book/doc_prog/index.html?macros.htm.

Костюк Д.А., Четверкина Г.А. Программирование на ассемблере в GNU/Linux: методическое пособие. Брест: изд-во БрГТУ, 2013. 68 с. Способ доступа: https://www.bstu.by/uploads/attachments/metodichki/kafedri/EVMiS_Assembler_v_GNU-Linux.pdf.

Столяров А.В. Программирование на языке ассемблера NASM для ОС Unix: Уч. Пособие. - М.: МАКС Пресс, 2011. - 188 с. Способ доступа: http://www.stolyarov.info/books/pdf/nasm_unix.pdf.

.

Приложение А. Основные арифметические команды ассемблера *NASM* для 32-х разрядной программы (синтаксис *Intel*)

1. Команда пересылки данных – пересылает число размером 1, 2 или 4 байта из источника в приемник:

`mov` Приемник, Источник

Допустимые варианты:

```
mov   reg, reg
mov   mem, reg
mov   reg, mem
mov   mem, imm
mov   reg, imm
mov   r/m16, sreg
mov   sreg, r/m16
```

Здесь и далее используются следующие условные обозначения:

reg – содержимое указанного регистра данных;

mem – содержимое памяти по указанному адресу;

imm – непосредственно заданный в команде операнд (литерал);

r/*m*16 – содержимое указанного 16-ти разрядного регистра или 16-ти разрядное содержимое памяти;

sreg – содержимое сегментного регистра.

Примечание. По правилам *NASM*, если операнд находится в памяти *mem*, то при указании адреса последний целиком записывается в квадратные скобки.

Примеры команд:

а) `mov  AX,BX`

б) `mov  ESI,1000`

в) `mov  [EDI],AL`

2. Команда перемещения и дополнения нулями **–** при перемещении значение источника помещается в младшие разряды, а в старшие заносятся нули. Формат команды:

movzx Приемник, Источник

Допустимые варианты:

```
movzx   r16/r32, r/m8
movzx   r32, r/m16
```

Примеры команд:

а) `movzx  EAX,BX`

б) `movzx  SI,AH`

3. Команда перемещения и дополнения знаковым разрядом **–** команда выполняется аналогично и имеет формат, совпадающий с форматом предыдущей команды, но в старшие разряды заносятся знаковые биты:

movsx Приемник, Источник

4. Команда обмена данных

XCHG Операнд1, Операнд 2

Допустимые варианты:

```
xchg    reg,reg
xchg    mem,reg
xchg    reg,mem
```

5-6. Команды записи слова или двойного слова в стек и извлечения из стека

```
PUSH    imm16 / imm32 / r16 / r32 / m16 / m32
POP     r16 / r32 / m16 / m32
```

Если в стек помещается 16-ти разрядное значение, то значение уменьшается на 2: *ESP*:= *ESP*–2, если помещается 32 разрядное значение, то – на 4: *ESP* := *ESP*–4.

Если из стека извлекается 16-ти разрядное значение, то значение увеличивается на 2: *ESP* := *ESP*+2, если помещается 32 разрядное значение, то – на 4: *ESP* := *ESP*+4.

Примеры:

```
push    SI
pop     word [EBX]
```

8-9. Команды сложения – складывает операнды, а результат помещает по адресу первого операнда. В отличие от *ADD* команда *ADC* добавляет к результату значение бита флага переноса *CF*.

ADD Операнд1, Операнд2

ADC Операнд1, Операнд2

Допустимые варианты:

```
add   reg,reg
add   mem,reg
add   reg,mem
add   mem,imm
add   reg,imm
```

10-11. Команды вычитания – вычитает из первого операнда второй и результат помещает по адресу первого операнда. В отличие от *SUB* команда *SBB* вычитает из результата значение бита флага переноса *CF*. Допустимые варианты те же, что и у сложения.

SUB Операнд1, Операнд2

SBB Операнд1, Операнд 2

13-14. Команды добавления/вычитания единицы

```
INC     reg/mem
DEC     reg/mem
```

Примеры:

```
inc     AX
dec    byte [EBX+EDI+8]
```

15-16. Команды умножения

```
MUL   <Операнд2>
IMUL  <Операнд2>
```

Допустимые варианты:

```
mul/imul   r|m8    ;  AX= AL*<Операнд2>
mul/imul   r|m16   ;  DX:AX= AX*<Операнд2>
mul/imul   r|m32   ;  EDX:EAX= EAX*<Операнд2>
```

В качестве второго операнда нельзя указать непосредственное значение!!!

Регистры первого операнда в команде не указываются. Местонахождение и длина результата операции зависит от размера второго операнда.

Пример:

```
mov    AX,4
imul    word [A]  ;   DX:AX:=AX*A
```

17-19. Команды «развертывания» чисел – операнды в команде не указываются. Операнд и его длина определяются кодом команды и не могут быть изменены. При выполнении команды происходит расширение записи числа до размера результата посредством размножения знакового разряда.

Команды часто используются при программировании деления чисел одинаковой размерности для обеспечения удвоенной длины делимого

```
CBW   ; байт в слово AL -> AX
CWD   ; слово в двойное слово AX -> DX:AX
CDQ          ; двойное слово в учетверенное EAX -> EDX:EAX
CWDE         ; слово в двойное слово AX -> EAX
```

20-21. Команды деления

DIV <Операнд2>

IDIV <Операнд2>

Допустимые варианты:

div/idiv r|m8 ; *AL*= *AX*:<Операнд2>, *AH* - остаток

div/idiv r|m16 ; *AX*= (*DX*:*AX*):<Операнд2>, *DX* - остаток

div/idiv r|m32 ; *EAX*= (*EDX*:*EAX*):<Операнд2>, *EDX* - остаток

В качестве второго операнда нельзя указать непосредственное значение!!!

Пример:

```
mov   AX,40
cwd
idiv    word [A]; AX:=(DX:AX):A
```

Приложение Б. Подпрограммы преобразования строки в число и числа в строку

Подпрограмма преобразования строки в число.

Ограничение: число должно находиться в интервале $-30000 \le x \le 30000$.

Вход: *ESI* – адрес строки, содержащей запись числа (положительные числа вводятся без знака), в конце введенной строки символ «10».

Выход: *EAX* – 32-х разрядное число, *EBX* – 0, если преобразование прошло без ошибок, и 1, если в процессе преобразования обнаружен ввод недопустимого символа или введенное число не попадает в заданный интервал.

```
StrToInt:
          push    edi
          mov     bh,'9'
          mov     bl,'0'
          push    esi       ; сохраняем адрес исходной строки
          cmp     byte[esi], '-'
          jne    .prod
          inc     esi       ; пропускаем знак минус
.prod     cld
          xor     di,di     ; обнуляем будущее число
.cycle:   lodsb             ; загружаем символ (цифру)
          cmp     al,10       ; если 10, то на конец
          je      .Return
          cmp     al,bl     ; сравниваем с кодом нуля
          jb      .Error    ; "ниже" – Ошибка
          cmp     al,bh     ; сравниваем с кодом девяти
          ja      .Error    ; "выше" – Ошибка
          sub     al,30h    ; получаем цифру из символа
          cbw               ; расширяем до слова
```

```
        push    ax          ; сохраняем в стеке
        mov     ax,10       ; заносим 10 в AX
        mul     di          ; умножаем, результат в DX:AX
        pop     di          ; в DI – очередная цифра
        add     ax,di
        mov     di,ax       ; в DI – накопленное число
        jmp     .cycle
.Return: pop    esi
        mov     ebx,0
        cmp     byte[esi],'-'
        jne     .J
        neg     di
.J      mov     ax,di
        cwde
        jmp     .R
.Error: pop     esi
        mov     eax,0
        mov     ebx,1
.R      pop     EDI
        ret
```

Подпрограмма преобразования числа в строку.

Ограничение: числа в интервале $-30000 \le x \le 30000$.

Вход: *EAX* – число, *ESI* – адрес области памяти для размещения строки результата (не менее 7 байт).

Выход: *EAX* – размер строки результата, запись числа будет прижата к левой границе области по адресу *ESI*, после числа будет вставлен символ с кодом 10.

```
IntToStr:
```

```
        push    edi
        push    ebx
        push    edx
        push    ecx
        push    esi
        mov     byte[esi],0  ; на место знака
        cmp     eax,0
        jge     .l1
        neg     eax
        mov     byte[esi],'-'
.l1     mov     byte[esi+6],10
        mov     edi,5
        mov     bx,10
.again: cwd                  ; расширили слово до двойного
        div     bx           ; делим результат на 10
        add     dl,30h  ; получаем из остатка код цифры
        mov     [esi+edi],DL  ; пишем символ в строку
        dec     edi          ; переводим указатель на
                             ; предыдущую позицию
        cmp     ax, 0        ; преобразовали все число?
        jne     .again
        mov     eax, 6
        sub     ecx, edi  ; длина результата+знак
        mov     eax,ecx
        inc     eax          ; длина результата+знак+0A
        inc     esi          ; пропускаем знак
        push    esi
        lea     esi,[esi+edi]  ; начало символов результата
        pop     edi
```

```
rep movsb
pop    esi
pop    ecx
pop    edx
pop    ebx
pop    edi
ret
```