

Отчёт по лабораторной работе №10

Понятие подпрограммы. Отладчик GDB

Аскеров Александр Эдуардович

Содержание

1	Цель работы	4
2	Выполнение лабораторной работы	5
2.1	Реализация подпрограмм в NASM	5
2.2	Отладка программ с помощью GDB	6
2.2.1	Добавление точек останова	10
2.2.2	Работа с данными программы в GDB	11
2.2.3	Обработка аргументов командной строки в GDB	14
2.3	Задания для самостоятельной работы	16
3	Выводы	19

Список иллюстраций

2.1	Создание каталога lab10 и файла lab10-1.asm	5
2.2	Проверка работы программы lab10-1.asm	5
2.3	Проверка работы программы lab10-1.asm (вывод $f(g(x))$)	6
2.4	Создание файла lab10-2.asm	6
2.5	Трансляция с ключом <code>-g</code>	6
2.6	Загрузка исполняемого файла в отладчик gdb	7
2.7	Проверим работу программы lab10-2.asm в оболочке GDB	7
2.8	Более подробный анализ программы с брейкпойнт	7
2.9	Дизассемблированный код программы	8
2.10	Переключение на отображение команд с Intel'овским синтаксисом	9
2.11	Включение режима псевдографики	9
2.12	Проверка наличия установки точки останова	10
2.13	Установка точки останова	10
2.14	Информация о всех установленных точках останова	11
2.15	Просмотр содержимого регистров	11
2.16	Просмотр значения msg1	12
2.17	Просмотр значения msg2	12
2.18	Изменение первого символа переменной msg1	12
2.19	Изменение первого символа переменной msg2	12
2.20	Вывод в различных форматах значения регистра edx	13
2.21	Изменение значения регистра ebx с помощью команды set	13
2.22	Завершение выполнения программы с помощью команды continue	14
2.23	Копирование lab9-2.asm в каталог lab10 под именем lab10-3.asm	14
2.24	Создание исполняемого файла	14
2.25	Загрузка исполняемого файла в отладчик с указыванием аргументов	14
2.26	Установка точки и её запуск	15
2.27	Адрес вершины стека	15
2.28	Просмотр остальных позиций стека	16
2.29	Результат работы программы lab10-4.asm	16
2.30	Проверка работы программы lab10-5.asm (неисправленной)	17
2.31	Работа с отладчиком	17
2.32	Проверка работы программы lab10-5.asm (исправленной)	18

1 Цель работы

Приобрести навыки написания программ с использованием подпрограмм. Познакомиться с методами отладки при помощи GDB и его основными возможностями.

2 Выполнение лабораторной работы

2.1 Реализация подпрограмм в NASM

1. Создадим каталог для выполнения лабораторной работы №10, перейдём в него и создадим файл lab10-1.asm.

```
[aeaskerov@fedora ~]$ mkdir ~/work/arch-pc/lab10  
[aeaskerov@fedora ~]$ cd ~/work/arch-pc/lab10  
[aeaskerov@fedora lab10]$ touch lab10-1.asm  
[aeaskerov@fedora lab10]$
```

Рис. 2.1: Создание каталога lab10 и файла lab10-1.asm

2. В качестве примера рассмотрим программу вычисления арифметического выражения $f(x) = 2x + 7$ с помощью подпрограммы `_calcul`. В данном примере `x` вводится с клавиатуры, а само выражение вычисляется в подпрограмме.

Внимательно изучим текст программы.

Введём в файл lab10-1.asm текст программы из листинга 10.1. Создадим исполняемый файл и проверим его работу.

```
[aeaskerov@fedora lab10]$ nasm -f elf lab10-1.asm  
[aeaskerov@fedora lab10]$ ld -m elf_i386 -o lab10-1 lab10-1.o  
[aeaskerov@fedora lab10]$ ./lab10-1  
Введите x: 2  
2x+7=11  
[aeaskerov@fedora lab10]$
```

Рис. 2.2: Проверка работы программы lab10-1.asm

Изменим текст программы, добавив подпрограмму `_subcalcul` в подпрограмму `_calcul` для вычисления выражения $f(g(x))$, где x вводится с клавиатуры, $f(x) = 2x + 7$, $g(x) = 3x - 1$. Т.е. x передаётся в подпрограмму `_calcul` из неё в подпрограмму `_subcalcul`, где вычисляется выражение $g(x)$, результат возвращается в `_calcul` и вычисляется выражение $f(g(x))$. Результат возвращается в основную программу для вывода результата на экран.

```
[aeaskerov@fedora lab10]$ nasm -f elf lab10-1.asm
[aeaskerov@fedora lab10]$ ld -m elf_i386 -o lab10-1 lab10-1.o
[aeaskerov@fedora lab10]$ ./lab10-1
Введите x: 2
2(3x-1)+7=17
[aeaskerov@fedora lab10]$
```

Рис. 2.3: Проверка работы программы `lab10-1.asm` (вывод $f(g(x))$)

2.2 Отладка программ с помощью GDB

Создадим файл `lab10-2.asm` с текстом программы из Листинга 10.2. (Программа печати сообщения `Hello world!`).

```
[aeaskerov@fedora lab10]$ touch lab10-2.asm
[aeaskerov@fedora lab10]$
```

Рис. 2.4: Создание файла `lab10-2.asm`

Получим исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом `-g`.

```
[aeaskerov@fedora lab10]$ nasm -f elf -g -l lab10-2.lst lab10-2.asm
[aeaskerov@fedora lab10]$ ld -m elf_i386 -o lab10-2 lab10-2.o
[aeaskerov@fedora lab10]$
```

Рис. 2.5: Трансляция с ключом `-g`

Загрузим исполняемый файл в отладчик gdb.

```
[aeaskerov@fedora lab10]$ gdb lab10-2
```

Рис. 2.6: Загрузка исполняемого файла в отладчик gdb

Проверим работу программы, запустив её в оболочке GDB с помощью команды `run` (сокращённо `r`).

```
(gdb) run
Starting program: /home/aeaskerov/work/arch-pc/lab10/lab10-2
Hello, world!
[Inferior 1 (process 4121) exited normally]
(gdb)
```

Рис. 2.7: Проверим работу программы lab10-2.asm в оболочке GDB

Для более подробного анализа программы установим брейкпойнт на метку `_start`, с которой начинается выполнение любой ассемблерной программы, и запустим её.

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10-2.asm, line 11.
(gdb) run
Starting program: /home/aeaskerov/work/arch-pc/lab10/lab10-2

Breakpoint 1, _start () at lab10-2.asm:11
11          mov eax,4
(gdb)
```

Рис. 2.8: Более подробный анализ программы с брейкпойнт

Посмотрим дизассемблированный код программы с помощью команды `disassemble`, начиная с метки `_start`.

```
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     $0x4,%eax
    0x08049005 <+5>:      mov     $0x1,%ebx
    0x0804900a <+10>:     mov     $0x804a000,%ecx
    0x0804900f <+15>:     mov     $0x8,%edx
    0x08049014 <+20>:     int     $0x80
    0x08049016 <+22>:     mov     $0x4,%eax
    0x0804901b <+27>:     mov     $0x1,%ebx
    0x08049020 <+32>:     mov     $0x804a008,%ecx
    0x08049025 <+37>:     mov     $0x7,%edx
    0x0804902a <+42>:     int     $0x80
    0x0804902c <+44>:     mov     $0x1,%eax
    0x08049031 <+49>:     mov     $0x0,%ebx
    0x08049036 <+54>:     int     $0x80
End of assembler dump.
(gdb) █
```

Рис. 2.9: Дизассемблированный код программы

Переключимся на отображение команд с Intel'овским синтаксисом, введя команду `set disassembly-flavor intel`.


```

(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     eax,0x4
    0x08049005 <+5>:      mov     ebx,0x1
    0x0804900a <+10>:     mov     ecx,0x804a000
    0x0804900f <+15>:     mov     edx,0x8
    0x08049014 <+20>:     int     0x80
    0x08049016 <+22>:     mov     eax,0x4
    0x0804901b <+27>:     mov     ebx,0x1
    0x08049020 <+32>:     mov     ecx,0x804a008
    0x08049025 <+37>:     mov     edx,0x7
    0x0804902a <+42>:     int     0x80
    0x0804902c <+44>:     mov     eax,0x1
    0x08049031 <+49>:     mov     ebx,0x0
    0x08049036 <+54>:     int     0x80
End of assembler dump.
(gdb) █

```

Рис. 2.10: Переключение на отображение команд с Intel'овским синтаксисом

Перечислим различия отображения синтаксиса машинных команд в режимах АТТ и Intel.

В АТТ сначала записывается адрес, потом регистр, перед адресом регистра ставится \$, перед названием регистра %. В Intel сначала регистр, потом адрес и перед ними ничего не ставится.

Включим режим псевдографики для более удобного анализа программы.

```

(gdb) layout asm
(gdb) layout regs

```

Рис. 2.11: Включение режима псевдографики

2.2.1 Добавление точек останова

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды – место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес.

Чтобы не было путаницы с номерами, перед адресом ставится “звёздочка”. На предыдущих шагах была установлена точка останова по имени метки (`_start`). Проверим это с помощью команды `info breakpoints` (кратко `i b`).

```
(gdb) info breakpoints
Num      Type             Disp Enb Address      What
1        breakpoint      keep y   0x08049000 lab10-2.asm:11
breakpoint already hit 1 time
(gdb)
```

Рис. 2.12: Проверка наличия установки точки останова

Установим ещё одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции. Определим адрес предпоследней инструкции (`mov ebx,0x0`) и установим точку останова.

```
(gdb) break *0x8049031
Breakpoint 2 at 0x8049031: file lab10-2.asm, line 24.
```

Рис. 2.13: Установка точки останова

Посмотрим информацию о всех установленных точках останова.

```
(gdb) i b
Num      Type          Disp Enb Address      What
1        breakpoint    keep y  0x08049000  lab10-2.asm:11
          breakpoint already hit 1 time
2        breakpoint    keep y  0x08049031  lab10-2.asm:24
(gdb) █
```

Рис. 2.14: Информация о всех установленных точках останова

2.2.2 Работа с данными программы в GDB

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Выполним 5 инструкций с помощью команды `stepi` (или `si`) и проследим за изменением значений регистров. Изменяются значения регистров `eax`, `ebx`, `ecx` и `edx`.

Посмотреть содержимое регистров также можно с помощью команды `info registers` (или `i r`).

```
native process 4178 In: _start                                L17   PC: 0x8049016
eax          0x8                                8
ecx          0x804a000                            134520832
edx          0x8                                8
ebx          0x1                                1
esp          0xffffd200                        0xffffd200
ebp          0x0                                0x0
--Type <RET> for more, q to quit, c to continue without paging-- █
```

Рис. 2.15: Просмотр содержимого регистров

Для отображения содержимого памяти можно использовать команду `x`, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: `x/NFU`.

С помощью команды `x &` также можно посмотреть содержимое переменной. Посмотрите значение переменной `msg1`.

```
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) █
```

Рис. 2.16: Просмотр значения msg1

Посмотрим значение переменной msg2 по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрим инструкцию mov esx,msg2 которая записывает в регистр esx адрес переменной msg2.

```
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb) █
```

Рис. 2.17: Просмотр значения msg2

Изменить значение для регистра или ячейки памяти можно с помощью команды set, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс \$, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си). Изменим первый символ переменной msg1.

```
(gdb) set {char}0x804a000='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) █
```

Рис. 2.18: Изменение первого символа переменной msg1

Заменяем любой символ во второй переменной msg2.

```
(gdb) set {char}0x804a008='S'
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "Sorld!\n\034"
(gdb) █
```

Рис. 2.19: Изменение первого символа переменной msg2

Чтобы посмотреть значения регистров используется команда `print /F val` (перед именем регистра обязательно ставится префикс `$`): `p/F $<регистр>`.

Выведем в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра `edx`.

```
(gdb) p/s $edx
$1 = 8
(gdb) p/x $edx
$2 = 0x8
(gdb) p/t $edx
$3 = 1000
(gdb) 
```

Рис. 2.20: Вывод в различных форматах значения регистра `edx`

С помощью команды `set` изменим значение регистра `ebx`.

```
(gdb) set $ebx='2'
(gdb) p/s $ebx
$4 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$5 = 2
(gdb) 
```

Рис. 2.21: Изменение значения регистра `ebx` с помощью команды `set`

Разница в выводе команд `p/s $ebx` заключается в том, что 50 – это номер символа 2 в таблице ASCII, в то время как 2 – это просто число 2.

Завершим выполнение программы с помощью команды `continue` (сокращенно `c`) или `stepi` (сокращенно `si`) и выйдем из GDB с помощью команды `quit` (сокращенно `q`).

```
(gdb) continue
Continuing.
Sorld!

Breakpoint 2, _start () at lab10-2.asm:24
(gdb) █
```

Рис. 2.22: Завершение выполнения программы с помощью команды continue

2.2.3 Обработка аргументов командной строки в GDB

Скопируем файл lab9-2.asm, созданный при выполнении лабораторной работы №9, с программой, выводящей на экран аргументы командной строки (Листинг 9.2), в файл с именем lab10-3.asm.

```
[aeaskerov@fedora lab10]$ cp ~/work/arch-pc/lab09/lab9-2.asm ~/work/arch-pc/lab10/lab10-3.asm
[aeaskerov@fedora lab10]$ █
```

Рис. 2.23: Копирование lab9-2.asm в каталог lab10 под именем lab10-3.asm

Создадим исполняемый файл.

```
[aeaskerov@fedora lab10]$ nasm -f elf -g -l lab10-3.lst lab10-3.asm
[aeaskerov@fedora lab10]$ ld -m elf_i386 -o lab10-3 lab10-3.o
[aeaskerov@fedora lab10]$ █
```

Рис. 2.24: Создание исполняемого файла

Для загрузки в gdb программы с аргументами необходимо использовать ключ `--args`. Загрузим исполняемый файл в отладчик, указав аргументы.

```
[aeaskerov@fedora lab10]$ gdb --args lab10-3 аргумент1 аргумент 2 'аргумент 3'
```

Рис. 2.25: Загрузка исполняемого файла в отладчик с указыванием аргументов

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек.

Исследуем расположение аргументов командной строки в стеке после запуска программы с помощью gdb.

Для начала установим точку останова перед первой инструкцией в программе и запустим её.

```
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab10-3.asm, line 7.
(gdb) run
Starting program: /home/aeaskerov/work/arch-pc/lab10/lab10-3 аргумент1 аргумент
2 аргумент\ 3

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.

Breakpoint 1, _start () at lab10-3.asm:7
7          pop ecx
(gdb) █
```

Рис. 2.26: Установка точки и её запуск

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы).

```
(gdb) x/x $esp
0xffffd1c0:      0x00000005
(gdb) █
```

Рис. 2.27: Адрес вершины стека

Как видно, число аргументов равно 5 – это имя программы lab10-3 и непосредственно аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’. Посмотрим остальные позиции стека – по адресу [esp + 4] располагается адрес в памяти, где находится имя программы, по адресу [esp + 8] хранится адрес первого аргумента, по адресу [esp + 12] – второго и т.д.

```

(gdb) x/x $esp
0xffffd1c0:      0x00000005
(gdb) x/s *(void**)($esp + 4)
0xffffd374:      "/home/aeaskerov/work/arch-pc/lab10/lab10-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd39f:      "аргумент1"
(gdb) x/s *(void**)($esp + 12)
0xffffd3b1:      "аргумент"
(gdb) x/s *(void**)($esp + 16)
0xffffd3c2:      "2"
(gdb) x/s *(void**)($esp + 20)
0xffffd3c4:      "аргумент 3"
(gdb) x/s *(void**)($esp + 24)
0x0:      <error: Cannot access memory at address 0x0>
(gdb) █

```

Рис. 2.28: Просмотр остальных позиций стека

Шаг равен четырём из-за того, что на позицию, которая хранит элемент стека, выделяется 4 байта.

2.3 Задания для самостоятельной работы

1. Преобразуем программу из лабораторной работы №9 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции $f(x)$ как подпрограмму.

```

[aeaskerov@fedora lab10]$ ./lab10-4 32 43 2
Функция:  $f(x)=8x-3$ 
Результат: 607
[aeaskerov@fedora lab10]$ ./lab10-4 55 666
Функция:  $f(x)=8x-3$ 
Результат: 5762
[aeaskerov@fedora lab10]$ █

```

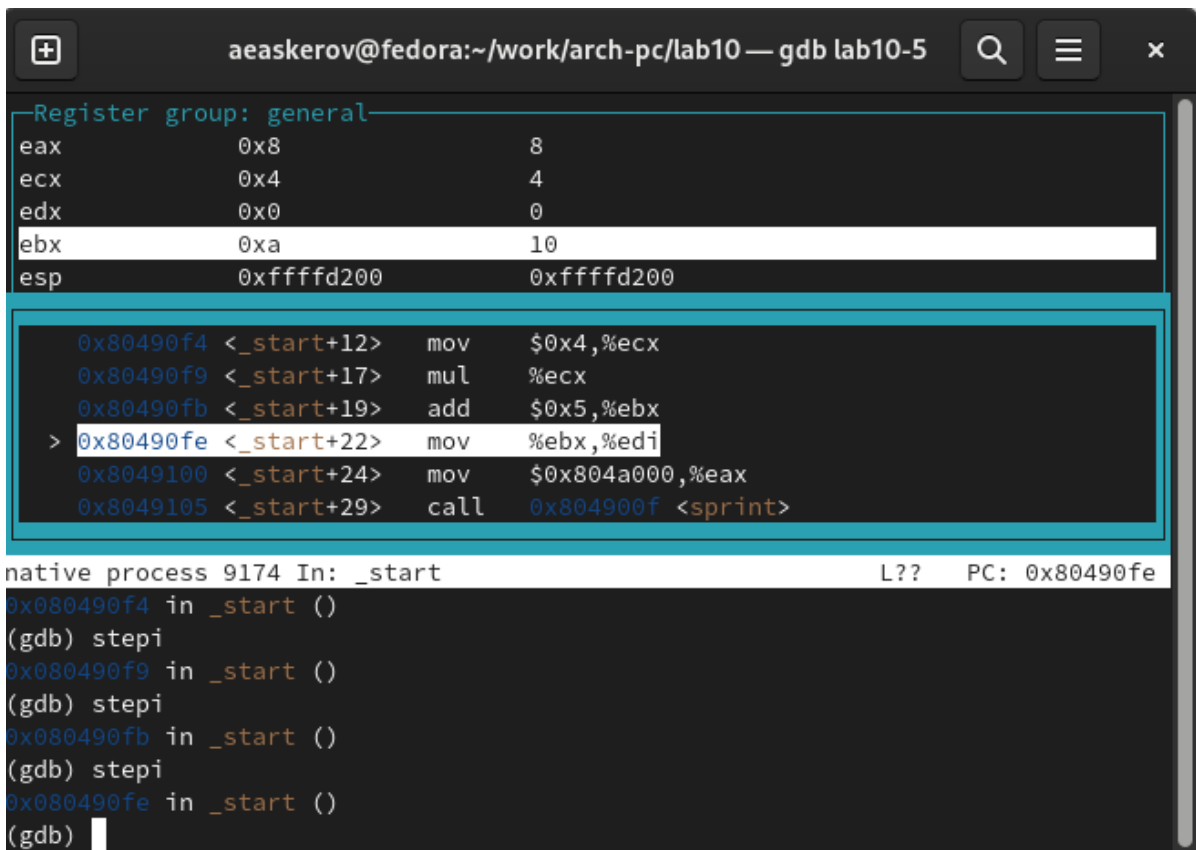
Рис. 2.29: Результат работы программы lab10-4.asm

2. В листинге 10.3 приведена программа вычисления выражения $(3 + 2) * 4 + 5$. При запуске данная программа даёт неверный результат. Проверим это. С помощью отладчика GDB, анализируя изменения значений регистров, определим ошибку и исправим её.

```
[aeaskerov@fedora lab10]$ nasm -f elf lab10-5.asm
[aeaskerov@fedora lab10]$ ld -m elf_i386 -o lab10-5 lab10-5.o
[aeaskerov@fedora lab10]$ ./lab10-5
Результат: 10
[aeaskerov@fedora lab10]$
```

Рис. 2.30: Проверка работы программы lab10-5.asm (неисправленной)

Результат действительно не соответствует ожидаемому.



```
aeaskerov@fedora:~/work/arch-pc/lab10 — gdb lab10-5
Register group: general
eax      0x8      8
ecx      0x4      4
edx      0x0      0
ebx      0xa      10
esp      0xffffd200 0xffffd200

0x80490f4 <_start+12> mov    $0x4,%ecx
0x80490f9 <_start+17> mul    %ecx
0x80490fb <_start+19> add    $0x5,%ebx
> 0x80490fe <_start+22> mov    %ebx,%edi
0x8049100 <_start+24> mov    $0x804a000,%eax
0x8049105 <_start+29> call   0x804900f <sprint>

native process 9174 In: _start L?? PC: 0x80490fe
0x80490f4 in _start ()
(gdb) stepi
0x80490f9 in _start ()
(gdb) stepi
0x80490fb in _start ()
(gdb) stepi
0x80490fe in _start ()
(gdb)
```

Рис. 2.31: Работа с отладчиком

С помощью GDB были пошагово просмотрены шаги выполнения программы.

Выяснилось, что регистр `ecx` умножается на изначальное значение регистра `eax`, а не на значение, полученное после сложения `eax` и `ebx`. Причина этого в том, что результат сложения сохраняется в `ebx`, в то время как `ecx` умножается на `eax`, который остался равным двум. Потом, вместо того чтобы прибавить 5 к `eax` (который должен был бы равняться 20, а равняется 2), 5 прибавляется к регистру `ebx`, равному пяти. Таким образом, мы получаем 8 и, впоследствии, 10, вместо 20 и, впоследствии, 25. Также нужно помещать в регистр `edi`, предназначенный для хранения ответа, регистр `eax`, а не регистр `ebx`.

Теперь исправим выявленные ошибки и получим верный ответ.

```
[aeaskerov@fedora lab10]$ nasm -f elf lab10-5.asm
[aeaskerov@fedora lab10]$ ld -m elf_i386 -o lab10-5 lab10-5.o
[aeaskerov@fedora lab10]$ ./lab10-5
Результат: 25
[aeaskerov@fedora lab10]$
```

Рис. 2.32: Проверка работы программы `lab10-5.asm` (исправленной)

3 Выводы

Приобретены навыки написания программ с использованием подпрограмм.
Изучены методы отладки при помощи GDB и его основные возможности.