

Отчёт по лабораторной работе 9

Понятие подпрограммы. Отладчик GDB.

Зиборова Вероника Николаевна НММбд-02-24

Содержание

1	Цель работы	5
2	Теоретическое введение	6
3	Выполнение лабораторной работы	7
3.1	Реализация подпрограмм в NASM	7
3.2	Отладка программ с помощью GDB	11
3.3	Задание для самостоятельной работы	23
4	Выводы	29
5	Ответы на вопросы	30

Список иллюстраций

3.1	Программа в файле lab9-1.asm	8
3.2	Запуск программы lab9-1.asm	9
3.3	Программа в файле lab9-1.asm	10
3.4	Запуск программы lab9-1.asm	11
3.5	Программа в файле lab9-2.asm	12
3.6	Запуск программы lab9-2.asm в отладчике	13
3.7	Дизассемблированный код	14
3.8	Дизассемблированный код в режиме интел	15
3.9	Точка остановки	16
3.10	Изменение регистров	17
3.11	Изменение регистров	18
3.12	Изменение значения переменной	19
3.13	Вывод значения регистра	20
3.14	Вывод значения регистра	21
3.15	Программа в файле lab9-3.asm	22
3.16	Вывод значения регистра	23
3.17	Программа в файле task-1.asm	24
3.18	Запуск программы task-1.asm	24
3.19	Код с ошибкой в файле task-2.asm	25
3.20	Отладка task-2.asm	26
3.21	Код исправлен в файле task-2.asm	27
3.22	Проверка работы task-2.asm	28

Список таблиц

1 Цель работы

Целью работы является приобретение навыков написания программ с использованием подпрограмм. Знакомство с методами отладки при помощи GDB и его основными возможностями.

2 Теоретическое введение

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

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

GDB (GNU Debugger — отладчик проекта GNU) работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки. Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя.

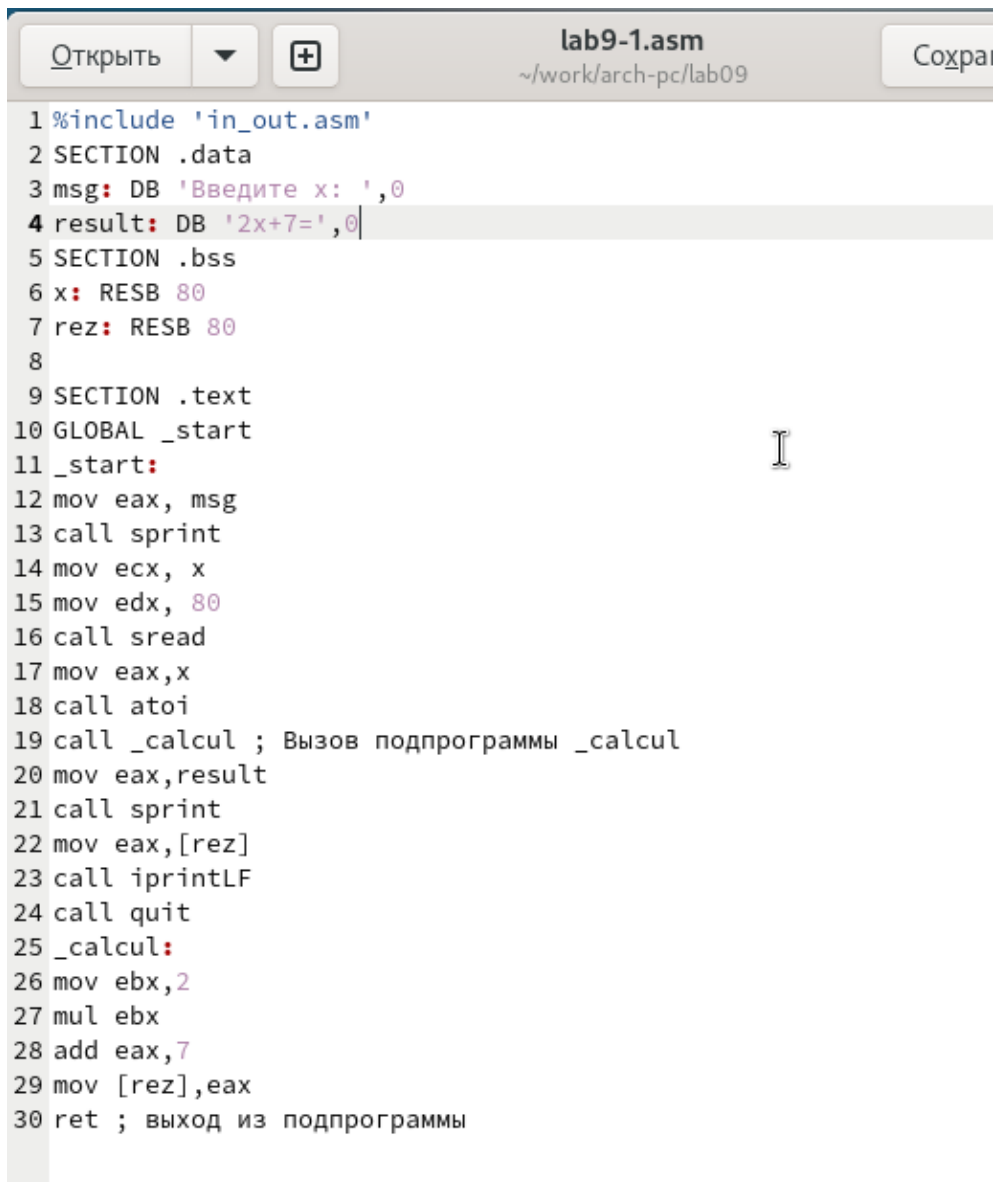
Подпрограмма — это, как правило, функционально законченный участок кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом

3 Выполнение лабораторной работы

3.1 Реализация подпрограмм в NASM

Я создала каталог для выполнения лабораторной работы №9 и перешла в него.

В качестве примера рассмотрим программу, которая вычисляет арифметическое выражение ($f(x) = 2x + 7$) с использованием подпрограммы `calcul`. В этом примере значение (x) вводится с клавиатуры, а само выражение вычисляется в подпрограмме.



```
1 %include 'in_out.asm'
2 SECTION .data
3 msg: DB 'Введите x: ',0
4 result: DB '2x+7=',0
5 SECTION .bss
6 x: RESB 80
7 rez: RESB 80
8
9 SECTION .text
10 GLOBAL _start
11 _start:
12 mov eax, msg
13 call sprint
14 mov ecx, x
15 mov edx, 80
16 call sread
17 mov eax,x
18 call atoi
19 call _calcul ; Вызов подпрограммы _calcul
20 mov eax,result
21 call sprint
22 mov eax,[rez]
23 call iprintLF
24 call quit
25 _calcul:
26 mov ebx,2
27 mul ebx
28 add eax,7
29 mov [rez],eax
30 ret ; выход из подпрограммы
```

Рис. 3.1: Программа в файле lab9-1.asm

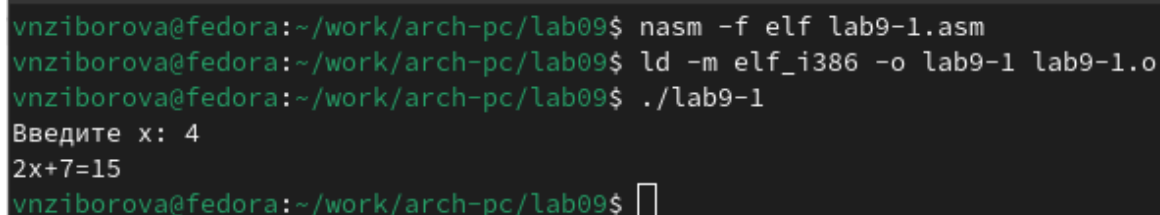
Первые строки программы отвечают за вывод сообщения на экран (с помощью вызова `sprint`), чтение данных, введенных с клавиатуры (с помощью вызова `sread`) и преобразование введенных данных из символьного вида в численный (с помощью вызова `atoi`).

После инструкции `call _calcul`, которая передает управление подпрограмме `_calcul`, будут выполнены инструкции, содержащиеся в подпрограмме.

Инструкция `ret` является последней в подпрограмме и её выполнение приводит

к возврату в основную программу к инструкции, следующей за инструкцией call, которая вызвала данную подпрограмму.

Последние строки программы реализуют вывод сообщения (с помощью вызова sprint), вывод результата вычисления (с помощью вызова iprintLF) и завершение программы (с помощью вызова quit).



```
vnziborova@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
vnziborova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
vnziborova@fedora:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 4
2x+7=15
vnziborova@fedora:~/work/arch-pc/lab09$
```

Рис. 3.2: Запуск программы lab9-1.asm

Я изменила текст программы, добавив подпрограмму subcalcul в подпрограмму calcul, для вычисления выражения $(f(g(x)))$, где (x) вводится с клавиатуры, $(f(x) = 2x + 7, g(x) = 3x - 1)$.

```
5
6 SECTION .bss
7 x: RESB 80
8 rez: RESB 80
9
10 SECTION .text
11 GLOBAL _start
12 _start:
13 mov eax, msg
14 call sprint
15 mov ecx, x
16 mov edx, 80
17 call sread
18 mov eax, x
19 call atoi
20 call _calcul ; Вызов подпрограммы _calcul
21 mov eax, result
22 call sprint
23 mov eax, [rez]
24 call iprintLF
25 call quit
26
27 _calcul:
28 call _subcalcul
29 mov ebx, 2
30 mul ebx
31 add eax, 7
32 mov [rez], eax
33 ret ; выход из подпрограммы
34
35 _subcalcul:
36 mov ebx, 3
37 mul ebx
38 sub eax, 1
39 ret
```

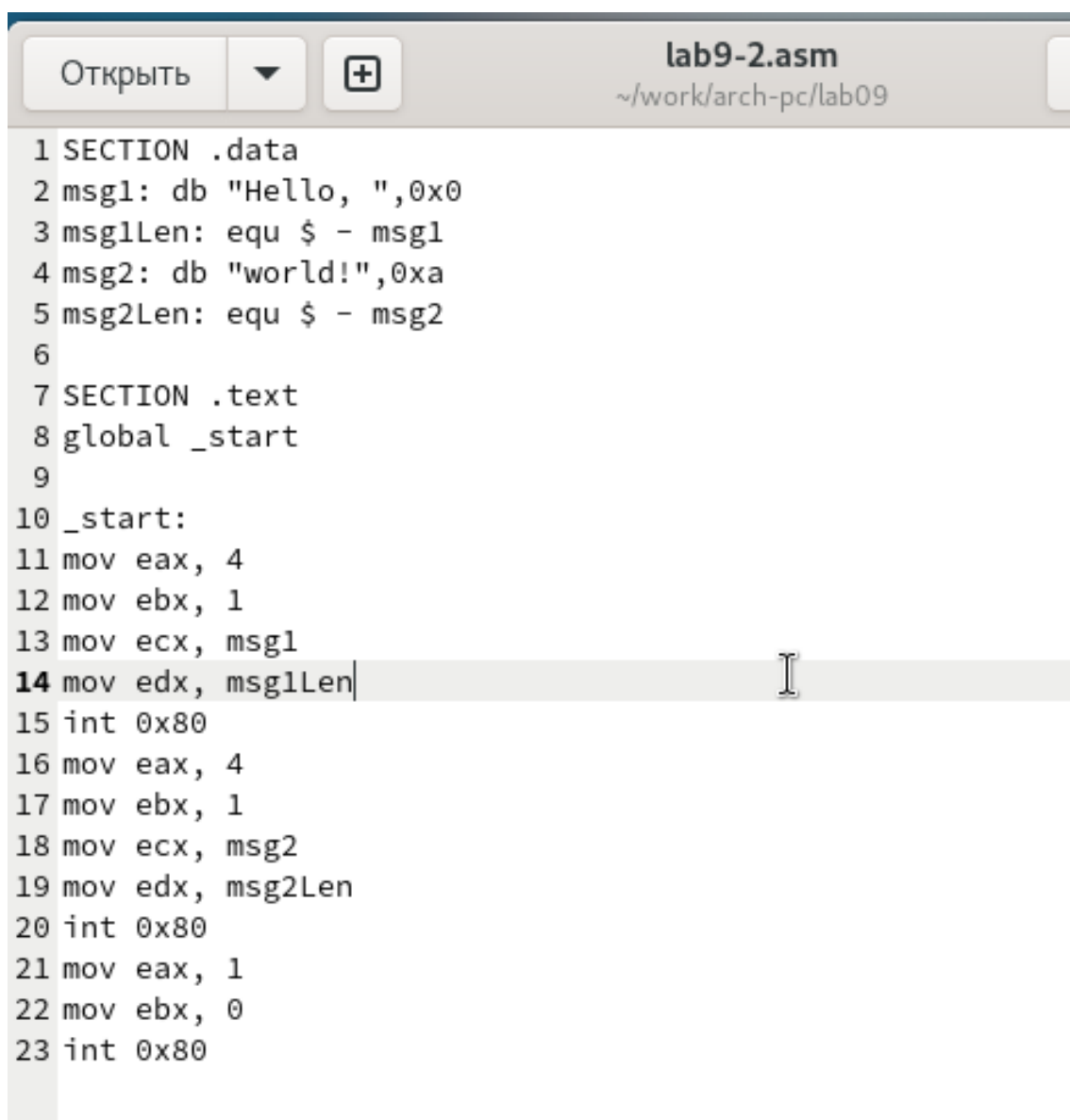
Рис. 3.3: Программа в файле lab9-1.asm

```
vnziborova@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
vnziborova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
vnziborova@fedora:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 4
2x+7=15
vnziborova@fedora:~/work/arch-pc/lab09$
vnziborova@fedora:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
vnziborova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
vnziborova@fedora:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 4
2(3x-1)+7=29
vnziborova@fedora:~/work/arch-pc/lab09$
```

Рис. 3.4: Запуск программы lab9-1.asm

3.2 Отладка программ с помощью GDB

Я создала файл lab9-2.asm с текстом программы из Листинга 9.2 (Программа печати сообщения Hello world!).



```
1 SECTION .data
2 msg1: db "Hello, ",0x0
3 msg1Len: equ $ - msg1
4 msg2: db "world!",0xa
5 msg2Len: equ $ - msg2
6
7 SECTION .text
8 global _start
9
10 _start:
11 mov eax, 4
12 mov ebx, 1
13 mov ecx, msg1
14 mov edx, msg1Len
15 int 0x80
16 mov eax, 4
17 mov ebx, 1
18 mov ecx, msg2
19 mov edx, msg2Len
20 int 0x80
21 mov eax, 1
22 mov ebx, 0
23 int 0x80
```

Рис. 3.5: Программа в файле lab9-2.asm

После того как я получила исполняемый файл, для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для чего трансляцию программ следует проводить с ключом `-g`.

Загрузила исполняемый файл в отладчик GDB и проверила работу программы, запустив её в оболочке GDB с помощью команды `run` (сокращенно `r`).

```

vnziborova@fedora:~/work/arch-pc/lab09$ nasm -f elf -g -l lab9-2.lst lab9-2.asm
vnziborova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
vnziborova@fedora:~/work/arch-pc/lab09$ gdb lab9-2
GNU gdb (Fedora Linux) 15.1-1.fc39
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab9-2...
(gdb) r
Starting program: /home/vnziborova/work/arch-pc/lab09/lab9-2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n])
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
Hello, world!
[Inferior 1 (process 8723) exited normally]
(gdb)

```

Рис. 3.6: Запуск программы lab9-2.asm в отладчике

Для более подробного анализа программы установила брейкпоинт на метку start, с которой начинается выполнение любой ассемблерной программы, и запустила её. Посмотрела дизассемблированный код программы.

```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2
(gdb) r
Starting program: /home/vnziborova/work/arch-pc/lab09/lab9-2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n])
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
Hello, world!
[Inferior 1 (process 8723) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab9-2.asm, line 11.
(gdb) r
Starting program: /home/vnziborova/work/arch-pc/lab09/lab9-2

Breakpoint 1, _start () at lab9-2.asm:11
11      mov eax, 4
(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. 3.7: Дизассемблированный код

```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2
(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) 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. 3.8: Дизассемблированный код в режиме интел

Для установки точки останова использовала команду `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать либо как номер строки программы (если есть исходный файл и программа компилировалась с отладочной информацией), либо как имя метки, или как адрес. Чтобы избежать путаницы с номерами, перед адресом ставится «звездочка».

На предыдущих шагах была установлена точка останова по имени метки `_start`. Проверила это с помощью команды `info breakpoints` (кратко `i b`). Затем устано-

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

```

vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2

Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd0f0 0xffffd0f0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049000 0x8049000 <_start>

B->0x8049000 <_start> mov eax,0x4
0x8049005 <_start+5> mov ebx,0x1
0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x8
0x8049014 <_start+20> int 0x80
0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a008
0x8049025 <_start+37> mov edx,0x7
0x804902a <_start+42> int 0x80

native process 8726 (asm) In: _start L11 PC: 0x8049000
(gdb) layout regs
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab9-2.asm, line 22.
(gdb) i b
Num    Type           Disp Enb Address      What
1      breakpoint      keep y   0x08049000 lab9-2.asm:11
       breakpoint already hit 1 time
2      breakpoint      keep y   0x08049031 lab9-2.asm:22
(gdb)

```

Рис. 3.9: Точка остановки

Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Я выполнила 5 инструкций с помощью команды stepi (или si) и проследила за изменением значений регистров.


```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2

Register group: general
eax      0x4      4
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd0f0 0xffffd0f0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049005 0x8049005 <_start+5>

B+ 0x8049000 <_start> mov eax,0x4
>0x8049005 <_start+5> mov ebx,0x1
0x804900a <_start+10> mov ecx,0x804a000
0x804900f <_start+15> mov edx,0x8
0x8049014 <_start+20> int 0x80
0x8049016 <_start+22> mov eax,0x4
0x804901b <_start+27> mov ebx,0x1
0x8049020 <_start+32> mov ecx,0x804a008
0x8049025 <_start+37> mov edx,0x7
0x804902a <_start+42> int 0x80

native process 8726 (asm) In: _start L12 PC: 0x8049005
eip      0x8049000 0x8049000 <_start>
eflags   0x202    [ IF ]
--Type <RET> for more, q to quit, c to continue without paging--
cs       0x23     35
ss       0x2b     43
ds       0x2b     43
es       0x2b     43
fs       0x0      0
gs       0x0      0
(gdb) si
(gdb) 
```

Рис. 3.10: Изменение регистров

```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd0f0 0xffffd0f0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>

B+ 0x8049000 <_start>    mov     eax,0x4
    0x8049005 <_start+5>  mov     ebx,0x1
    0x804900a <_start+10> mov     ecx,0x804a000
    0x804900f <_start+15> mov     edx,0x8
    0x8049014 <_start+20> int     0x80
> 0x8049016 <_start+22> mov     eax,0x4
    0x804901b <_start+27> mov     ebx,0x1
    0x8049020 <_start+32> mov     ecx,0x804a008
    0x8049025 <_start+37> mov     edx,0x7
    0x804902a <_start+42> int     0x80

native process 8726 (asm) In: _start L16 PC: 0x8049016
ss      0x2b      43
ds      0x2b      43
es      0x2b      43
fs      0x0      0
gs      0x0      0
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) 
```

Рис. 3.11: Изменение регистров

Посмотрела значение переменной `msg1` по имени и значение переменной `msg2` по адресу.

Изменить значение для регистра или ячейки памяти можно с помощью команды `set`, указав имя регистра или адрес. Я изменила первый символ переменной `msg1`.

```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd0f0 0xffffd0f0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049016 0x8049016 <_start+22>

B+ 0x8049000 <_start>    mov    eax,0x4
    0x8049005 <_start+5>  mov    ebx,0x1
    0x804900a <_start+10> mov    ecx,0x804a000
    0x804900f <_start+15> mov    edx,0x8
    0x8049014 <_start+20> int     0x80
>0x8049016 <_start+22>  mov    eax,0x4
    0x804901b <_start+27> mov    ebx,0x1
    0x8049020 <_start+32> mov    ecx,0x804a008
    0x8049025 <_start+37> mov    edx,0x7
    0x804902a <_start+42> int     0x80

native process 8726 (asm) In: _start      L16  PC: 0x8049016
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) set {char}0x804a008='L'
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "Lorld!\n\034"
(gdb) 
```

Рис. 3.12: Изменение значения переменной

Я вывела значение регистра `edx` в различных форматах (в шестнадцатеричном, двоичном и символьном).

The screenshot shows a GDB terminal window with the title bar 'vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2'. The 'Register group: general' window is open, displaying the following register values:

Register	Value (hex)	Value (decimal)
eax	0x8	8
ecx	0x804a000	134520832
edx	0x8	8
ebx	0x1	1
esp	0xffffd0f0	0xffffd0f0
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x8049016	0x8049016 <_start+22>

Below the register window, the assembly code is displayed, with the instruction at address 0x8049016 highlighted:

```
B+ 0x8049000 <_start>    mov     eax,0x4
0x8049005 <_start+5>    mov     ebx,0x1
0x804900a <_start+10>   mov     ecx,0x804a000
0x804900f <_start+15>   mov     edx,0x8
0x8049014 <_start+20>   int     0x80
>0x8049016 <_start+22>  mov     eax,0x4
0x804901b <_start+27>   mov     ebx,0x1
0x8049020 <_start+32>   mov     ecx,0x804a008
0x8049025 <_start+37>   mov     edx,0x7
0x804902a <_start+42>   int     0x80
```

The command prompt shows the following commands and their outputs:

```
(gdb) p/s $ecx
$3 = 134520832
(gdb) p/x $ecx
$4 = 0x804a000
(gdb) p/s $edx
$5 = 8
(gdb) p/t $edx
$6 = 1000
(gdb) p/x $edx
$7 = 0x8
(gdb)
```

Рис. 3.13: Вывод значения регистра

С помощью команды set изменила значение регистра ebx.

The screenshot shows a GDB terminal window with the title bar "vnziborova@fedora:~/work/arch-pc/lab09 — gdb lab9-2". The window is divided into two main sections. The top section, titled "Register group: general", displays the values of general-purpose registers:

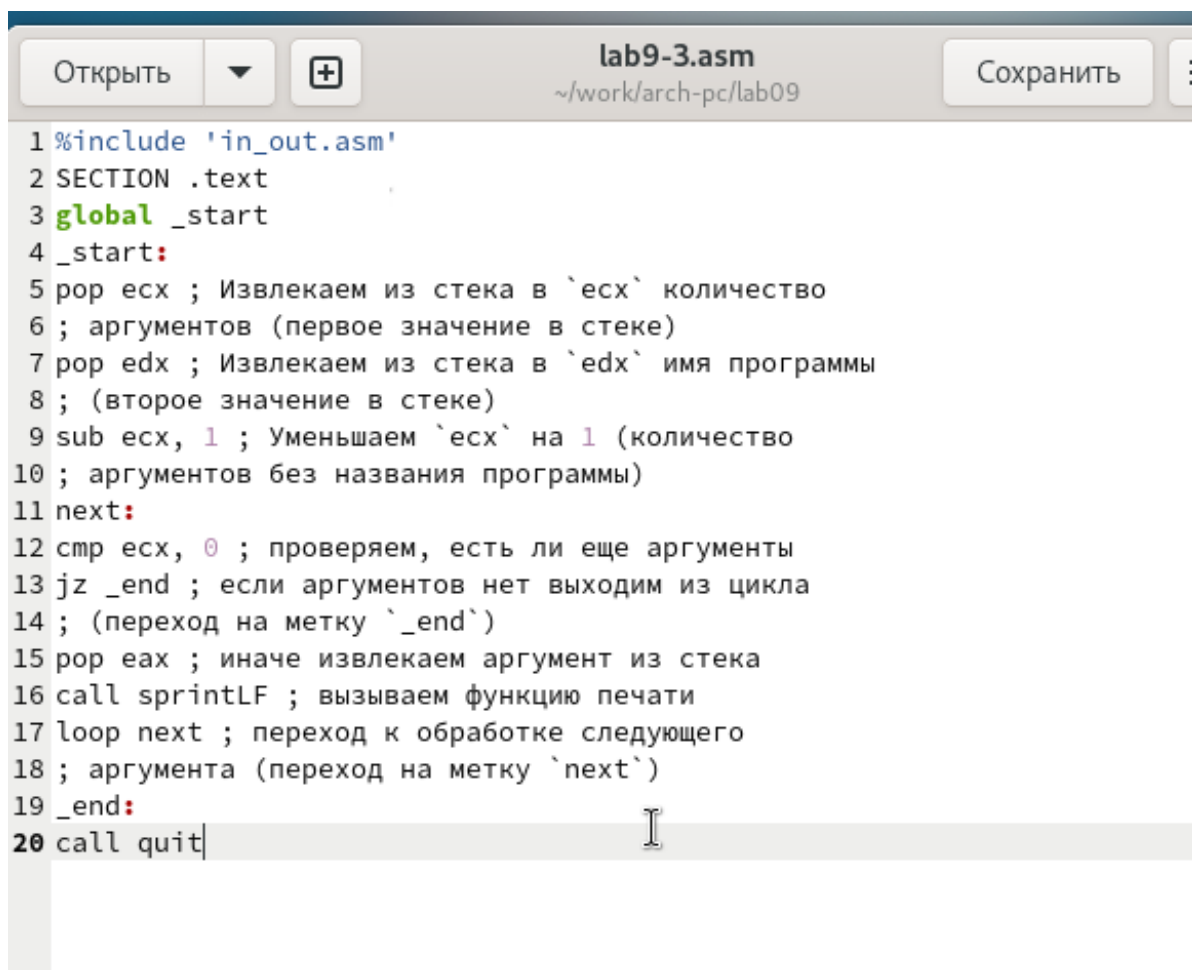
Register	Value (hex)	Value (dec)
eax	0x8	8
ecx	0x804a000	134520832
edx	0x8	8
ebx	0x2	2
esp	0xffffd0f0	0xffffd0f0
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x8049016	0x8049016 <_start+22>

. The bottom section displays assembly code starting from address 0x8049000. The instruction at 0x8049016, "mov eax,0x4", is highlighted. Below the assembly view, the GDB command prompt shows a series of commands and their outputs:

```
(gdb) p/t $edx
$6 = 1000
(gdb) p/x $edx
$7 = 0x8
(gdb) set $ebx='2'
(gdb) p/s $ebx
$8 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$9 = 2
(gdb) 
```

Рис. 3.14: Вывод значения регистра

Я скопировала файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой, выводящей на экран аргументы командной строки. Создала исполняемый файл. Для загрузки программы с аргументами в GDB необходимо использовать ключ `-args`. Загрузила исполняемый файл в отладчик, указав аргументы.



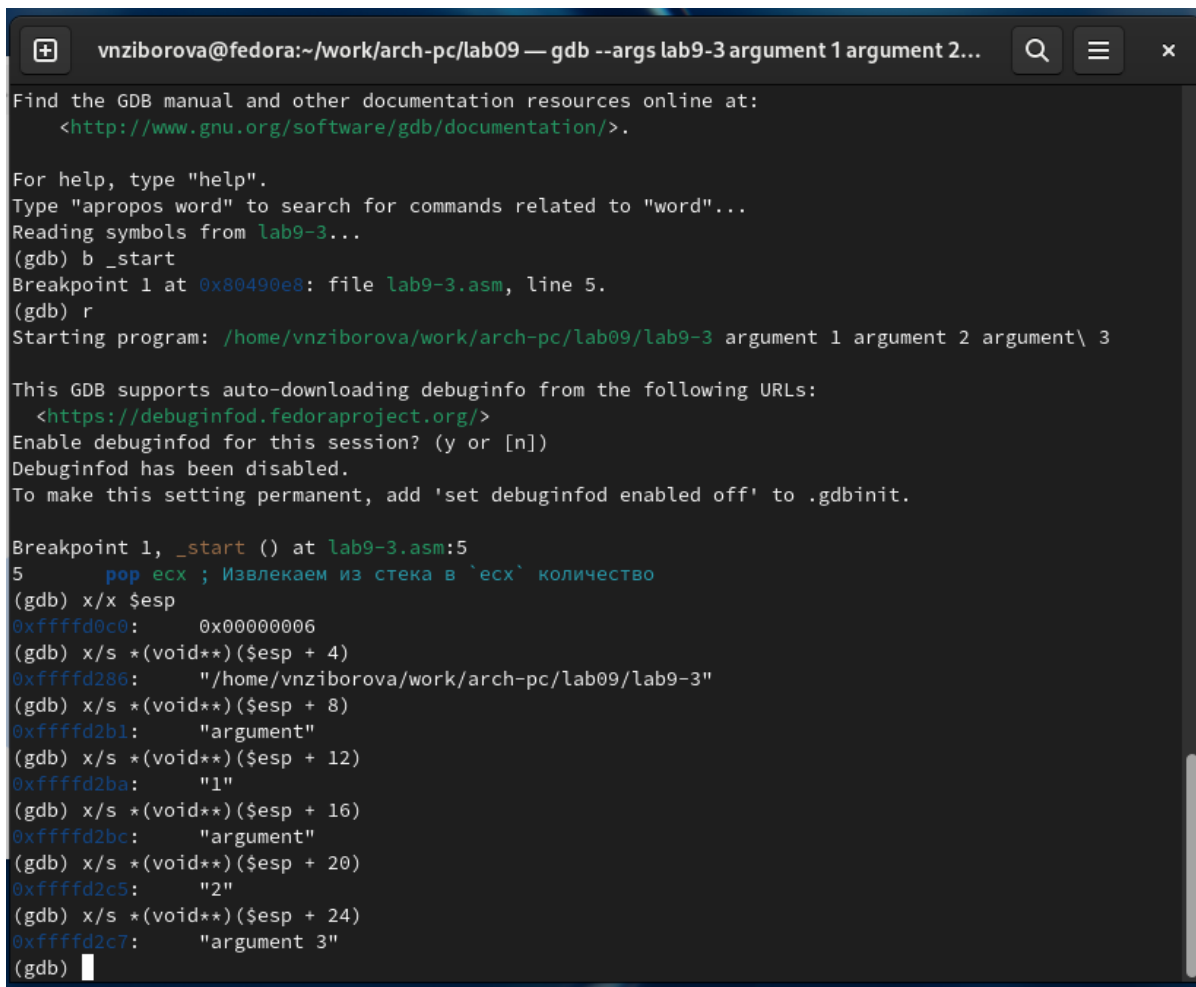
```
1 %include 'in_out.asm'
2 SECTION .text
3 global _start
4 _start:
5 pop ecx ; Извлекаем из стека в `ecx` количество
6 ; аргументов (первое значение в стеке)
7 pop edx ; Извлекаем из стека в `edx` имя программы
8 ; (второе значение в стеке)
9 sub ecx, 1 ; Уменьшаем `ecx` на 1 (количество
10 ; аргументов без названия программы)
11 next:
12 cmp ecx, 0 ; проверяем, есть ли еще аргументы
13 jz _end ; если аргументов нет выходим из цикла
14 ; (переход на метку `_end`)
15 pop eax ; иначе извлекаем аргумент из стека
16 call sprintf ; вызываем функцию печати
17 loop next ; переход к обработке следующего
18 ; аргумента (переход на метку `next`)
19 _end:
20 call quit
```

Рис. 3.15: Программа в файле lab9-3.asm

Для начала установила точку останова перед первой инструкцией в программе и запустила её.

Адрес вершины стека хранится в регистре `esp`, и по этому адресу располагается число, равное количеству аргументов командной строки (включая имя программы). Как видно, число аргументов равно 5 — это имя программы `lab9-3` и непосредственно аргументы: `аргумент1`, `аргумент2` и `аргумент3`.

Посмотрела остальные позиции стека — по адресу `[esp+4]` располагается адрес в памяти, где находится имя программы, по адресу `[esp+8]` — адрес первого аргумента, по адресу `[esp+12]` — второго и т.д.



```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb --args lab9-3 argument 1 argument 2...
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab9-3...
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab9-3.asm, line 5.
(gdb) r
Starting program: /home/vnziborova/work/arch-pc/lab09/lab9-3 argument 1 argument 2 argument\ 3

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

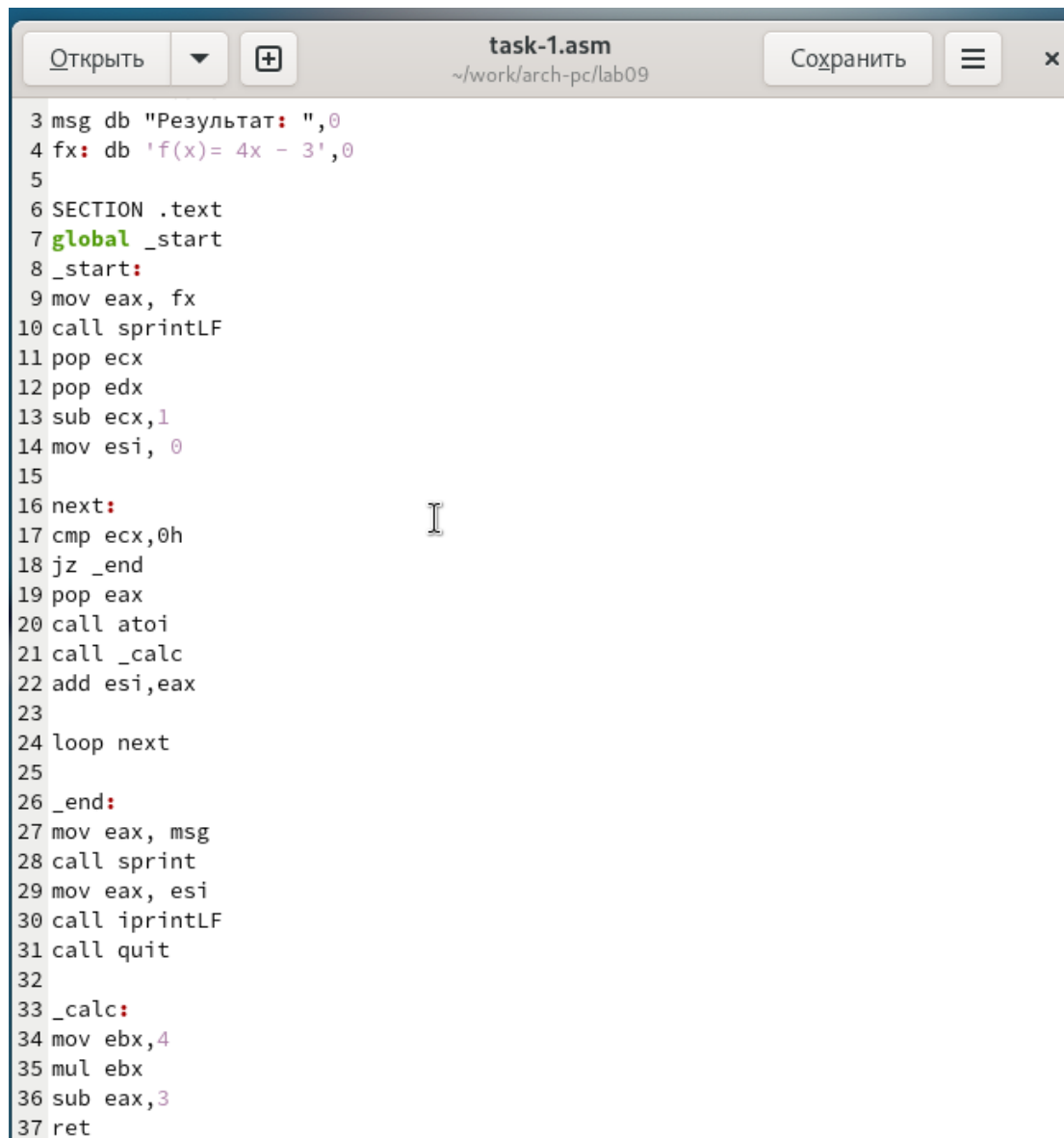
Breakpoint 1, _start () at lab9-3.asm:5
5      pop ecx ; Извлекаем из стека в `ecx` количество
(gdb) x/x $esp
0xffffd0c0: 0x00000006
(gdb) x/s *(void**)(esp + 4)
0xffffd286: "/home/vnziborova/work/arch-pc/lab09/lab9-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd2b1: "argument"
(gdb) x/s *(void**)(esp + 12)
0xffffd2ba: "1"
(gdb) x/s *(void**)(esp + 16)
0xffffd2bc: "argument"
(gdb) x/s *(void**)(esp + 20)
0xffffd2c5: "2"
(gdb) x/s *(void**)(esp + 24)
0xffffd2c7: "argument 3"
(gdb) 
```

Рис. 3.16: Вывод значения регистра

Объяснила, почему шаг изменения адреса равен 4 ([esp+4], [esp+8], [esp+12]) — шаг равен размеру переменной (4 байта).

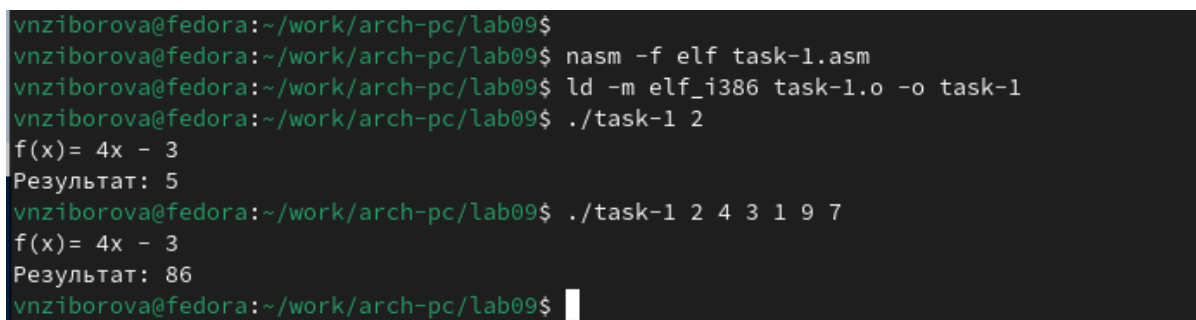
3.3 Задание для самостоятельной работы

Я переписала программу из лабораторной работы №8, чтобы вычислить значение функции ($f(x)$) в виде подпрограммы.



```
3 msg db "Результат: ",0
4 fx: db 'f(x)= 4x - 3',0
5
6 SECTION .text
7 global _start
8 _start:
9 mov eax, fx
10 call sprintf
11 pop ecx
12 pop edx
13 sub ecx,1
14 mov esi, 0
15
16 next:
17 cmp ecx,0h
18 jz _end
19 pop eax
20 call atoi
21 call _calc
22 add esi,eax
23
24 loop next
25
26 _end:
27 mov eax, msg
28 call sprintf
29 mov eax, esi
30 call iprintLF
31 call quit
32
33 _calc:
34 mov ebx,4
35 mul ebx
36 sub eax,3
37 ret
```

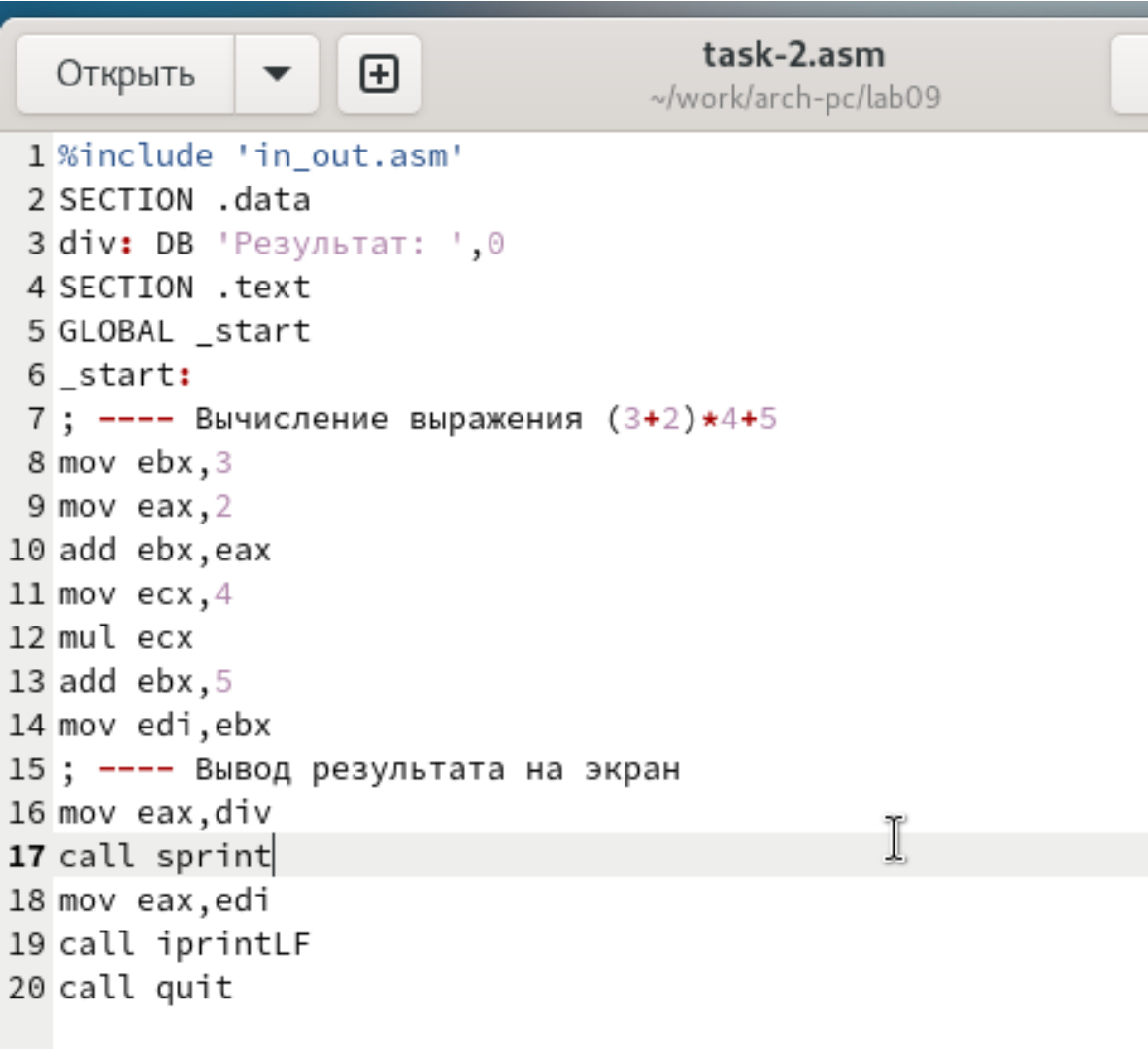
Рис. 3.17: Программа в файле task-1.asm



```
vnziborova@fedora:~/work/arch-pc/lab09$
vnziborova@fedora:~/work/arch-pc/lab09$ nasm -f elf task-1.asm
vnziborova@fedora:~/work/arch-pc/lab09$ ld -m elf_i386 task-1.o -o task-1
vnziborova@fedora:~/work/arch-pc/lab09$ ./task-1 2
f(x)= 4x - 3
Результат: 5
vnziborova@fedora:~/work/arch-pc/lab09$ ./task-1 2 4 3 1 9 7
f(x)= 4x - 3
Результат: 86
vnziborova@fedora:~/work/arch-pc/lab09$
```

Рис. 3.18: Запуск программы task-1.asm

Приведенный ниже листинг программы вычисляет выражение $(3+2)*4+5$. Однако при запуске программа дает неверный результат. Я проверила это и решила использовать отладчик GDB для анализа изменений значений регистров и определения ошибки.



```
task-2.asm
~/work/arch-pc/lab09

1 %include 'in_out.asm'
2 SECTION .data
3 div: DB 'Результат: ',0
4 SECTION .text
5 GLOBAL _start
6 _start:
7 ; ---- Вычисление выражения (3+2)*4+5
8 mov ebx,3
9 mov eax,2
10 add ebx,eax
11 mov ecx,4
12 mul ecx
13 add ebx,5
14 mov edi,ebx
15 ; ---- Вывод результата на экран
16 mov eax,div
17 call sprint
18 mov eax,edi
19 call iprintLF
20 call quit
```

Рис. 3.19: Код с ошибкой в файле task-2.asm

The screenshot shows a GDB window titled "vnziborova@fedora:~/work/arch-pc/lab09 — gdb task-2". The "Register group: general" section displays the following values:

Register	Value	Comment
eax	0x8	8
ecx	0x4	4
edx	0x0	0
ebx	0xa	10
esp	0xffffd0f0	0xffffd0f0
ebp	0x0	0x0
esi	0x0	0
edi	0xa	10
eip	0x8049100	0x8049100 <_start+24>

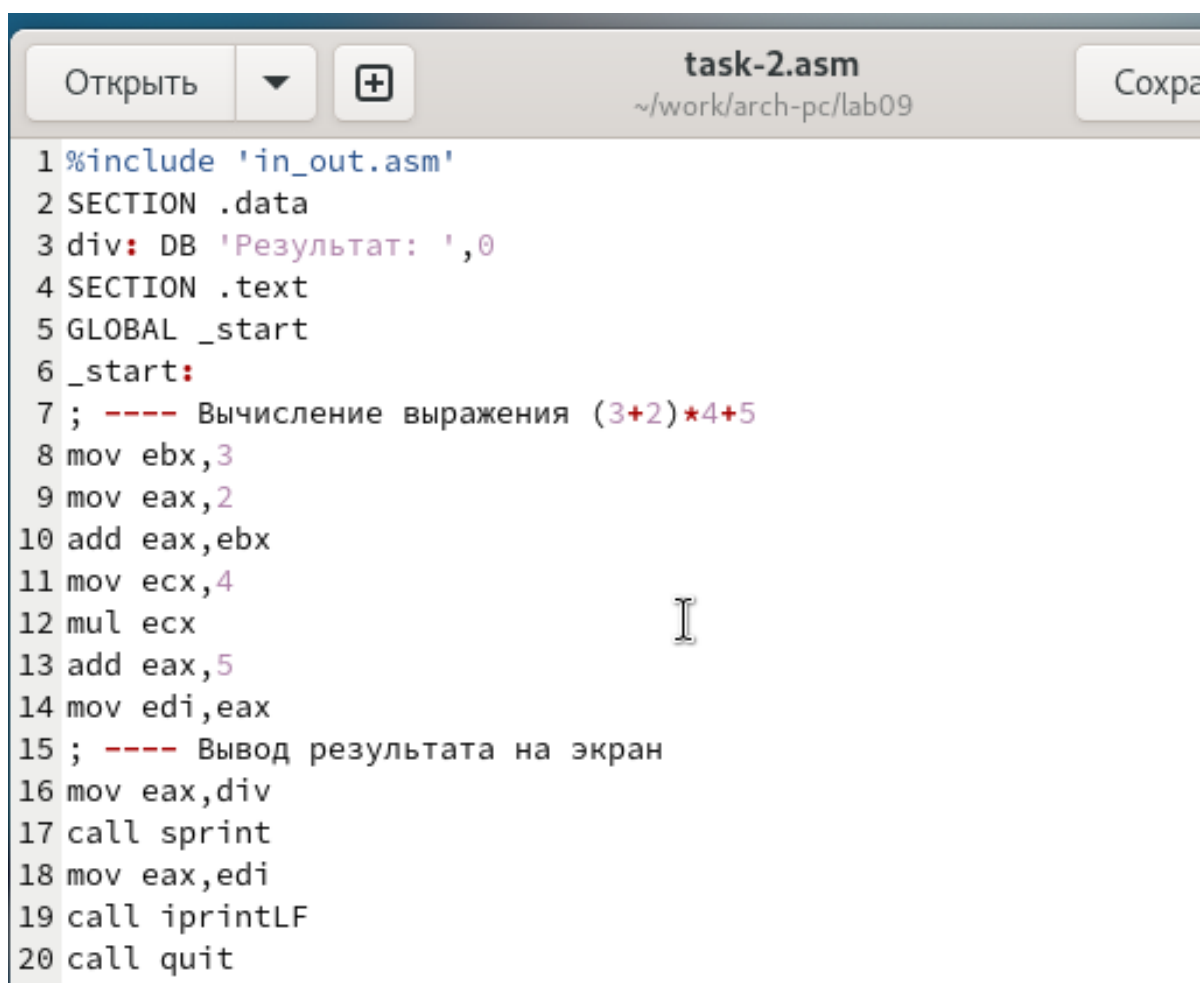
The assembly window shows the following code:

```
B+ 0x80490e8 <_start>    mov     ebx,0x3
0x80490ed <_start+5>    mov     eax,0x2
0x80490f2 <_start+10>   add     ebx,eax
0x80490f4 <_start+12>   mov     ecx,0x4
0x80490f9 <_start+17>   mul     ecx
0x80490fb <_start+19>   add     ebx,0x5
0x80490fe <_start+22>   mov     edi,ebx
>0x8049100 <_start+24> mov     eax,0x804a000
0x8049105 <_start+29>   call    0x804900f <sprint>
0x804910a <_start+34>   mov     eax,edi
```

The status bar indicates "native process 8860 (asm) In: _start L16 PC: 0x8049100". Below the assembly window, a message states: "To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit." The breakpoint list shows "Breakpoint 1, _start () at task-2.asm:8". The command window contains several "(gdb) si" commands.

Рис. 3.20: Отладка task-2.asm

Я заметила, что порядок аргументов в инструкции add был перепутан, и что при завершении работы вместо eax значение отправлялось в edi. Вот исправленный код программы:



```
1 %include 'in_out.asm'
2 SECTION .data
3 div: DB 'Результат: ',0
4 SECTION .text
5 GLOBAL _start
6 _start:
7 ; ---- Вычисление выражения (3+2)*4+5
8 mov ebx,3
9 mov eax,2
10 add eax,ebx
11 mov ecx,4
12 mul ecx
13 add eax,5
14 mov edi,eax
15 ; ---- Вывод результата на экран
16 mov eax,div
17 call sprint
18 mov eax,edi
19 call iprintLF
20 call quit
```

Рис. 3.21: Код исправлен в файле task-2.asm

```
vnziborova@fedora:~/work/arch-pc/lab09 — gdb task-2

Register group: general
eax      0x19      25
ecx      0x4       4
edx      0x0       0
ebx      0x3       3
esp      0xffffd0f0 0xffffd0f0
ebp      0x0       0x0
esi      0x0       0
edi      0x19      25
eip      0x8049100 0x8049100 <_start+24>

B+ 0x80490e8 <_start>    mov     ebx,0x3
0x80490ed <_start+5>    mov     eax,0x2
0x80490f2 <_start+10>   add     eax,ebx
0x80490f4 <_start+12>   mov     ecx,0x4
0x80490f9 <_start+17>   mul     ecx
0x80490fb <_start+19>   add     eax,0x5
0x80490fe <_start+22>   mov     edi,eax
>0x8049100 <_start+24> mov     eax,0x804a000
0x8049105 <_start+29>   call   0x804900f <sprint>
0x804910a <_start+34>   mov     eax,edi

native process 8907 (asm) In: _start L16 PC: 0x8049100
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.

Breakpoint 1, _start () at task-2.asm:8
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
```

Рис. 3.22: Проверка работы task-2.asm

4 Выводы

Освоили работу с подпрограммами и отладчиком.

5 Ответы на вопросы

1. Какие языковые средства используются в ассемблере для оформления и активизации подпрограмм?

В ассемблере для оформления и активизации подпрограмм используются такие средства, как:

- **Метки (labels)** — служат для обозначения начала подпрограммы.
- **Инструкция call** — активирует вызов подпрограммы, передавая управление на указанную метку или адрес.
- **Инструкция ret** — возвращает управление в точку, где была вызвана подпрограмма.

2. Объясните механизм вызова подпрограмм.

Механизм вызова подпрограммы состоит в следующем:

- При выполнении инструкции call происходит сохранение адреса следующей команды (адрес возврата) на стеке.
- Управление передается на начало подпрограммы, где выполняются её инструкции.
- После завершения работы подпрограммы выполняется инструкция ret, которая извлекает адрес возврата из стека и передает управление обратно в вызывающую программу.

3. Как используется стек для обеспечения взаимодействия между вызывающей и вызываемой процедурами?

Стек используется для:

- Сохранения адреса возврата (то есть места, куда программа должна вернуться после завершения подпрограммы).
- Хранения значений регистров и локальных переменных, если подпрограмма изменяет их, чтобы сохранить состояние вызывающей программы.

4. Каково назначение операнда в команде `ret`?

Операнд в команде `ret` обычно указывает на количество байт, которые нужно очистить из стека после возврата из подпрограммы. Это нужно для того, чтобы сбросить параметры, переданные подпрограмме через стек. Без операнда `ret` по умолчанию извлекает адрес возврата из стека и передает управление туда.

5. Для чего нужен отладчик?

Отладчик (например, GDB) используется для:

- Тщательного анализа работы программы на каждом этапе.
- Поиска и устранения ошибок (bugfixing).
- Проверки содержимого регистров, памяти, стеков и переменных во время выполнения программы.
- Управления выполнением программы (пауза, пошаговое выполнение, установка точек останова).

6. Объясните назначение отладочной информации и как нужно компилировать программу, чтобы в ней присутствовала отладочная информация.

Отладочная информация помогает отладчику отслеживать исходный код программы и сопоставлять его с машинным кодом. Чтобы включить отладочную информацию, программу нужно компилировать с использованием

флага -g в командной строке компилятора. Это добавляет метки, номера строк и другую информацию о исходном коде в исполняемый файл.

7. **Расшифруйте и объясните следующие термины: breakpoint, watchpoint, checkpoint, catchpoint и call stack.**

- **Breakpoint** — это точка останова, на которой выполнение программы приостанавливается. Обычно устанавливается на строке исходного кода или на конкретной инструкции.
- **Watchpoint** — это точка останова, которая срабатывает, когда изменяется значение определённой переменной или памяти.
- **Checkpoint** — точка сохранения состояния программы, к которой можно вернуться при отладке.
- **Catchpoint** — это точка останова, которая срабатывает при возникновении определённых событий, например, при исключениях или сигналах.
- **Call stack** — стек вызовов, который хранит информацию о последовательности вызова подпрограмм, а также о локальных переменных и адресах возврата.

8. **Назовите основные команды отладчика gdb и как они могут быть использованы для отладки программ.**

- **run (r)** — запуск программы в отладчике.
- **break (b)** — установка точки останова на метке или строке программы.
- **next (n)** — выполнение следующей строки исходного кода (без захода в подпрограмму).
- **step (s)** — выполнение следующей строки исходного кода, с заходом в подпрограмму.
- **continue (c)** — продолжение выполнения программы после остановки на точке останова.
- **print (p)** — вывод значения переменной или выражения.

- **info locals** — вывод значений локальных переменных в текущем контексте.
- **backtrace (bt)** — вывод стека вызовов, показывающий последовательность вызовов подпрограмм.
- **quit (q)** — завершение работы отладчика.