

# **Лабораторная работа №10**

**Понятие подпрограммы. Отладчик GDB**

Татьяна Алексеевна Коннова, НПИбд-01-22

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>4</b>
1.1	Задание . . . . .	4
1.2	Выполнение лабораторной работы . . . . .	4
1.2.1	Реализация подпрограмм в NASM . . . . .	4
1.3	9.3.2. Отладка программ с помощью GDB . . . . .	7
1.4	10.4.2.1. Добавление точек останова . . . . .	13
1.5	10.4.2.2. Работа с данными программы в GDB . . . . .	16
1.6	10.4.2.3. Обработка аргументов командной строки в GDB . . . . .	20
<b>2</b>	<b>10.5. Самостоятельная работа</b>	<b>23</b>
<b>3</b>	<b>Выводы</b>	<b>26</b>

# Список иллюстраций

1.1	lab10_1.asm . . . . .	5
1.2	subcalcul . . . . .	7
1.3	lab10-2.asm . . . . .	8
1.4	lab10-2.asm . . . . .	9
1.5	breakpoint . . . . .	11
1.6	layout . . . . .	13
1.7	intel change . . . . .	15
1.8	intel change . . . . .	16
1.9	intel change . . . . .	18
1.10	change . . . . .	19
1.11	change . . . . .	19
1.12	лаб10_3 . . . . .	20
1.13	gdb . . . . .	21
1.14	gdb . . . . .	22
2.1	лаб10_5 . . . . .	23
2.2	gdb . . . . .	24
2.3	лаб10_5 . . . . .	24
2.4	Правки в работе . . . . .	25

# 1 Цель работы

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

## 1.1 Задание

Знакомство с подпрограммами

## 1.2 Выполнение лабораторной работы

### 1.2.1 Реализация подпрограмм в NASM

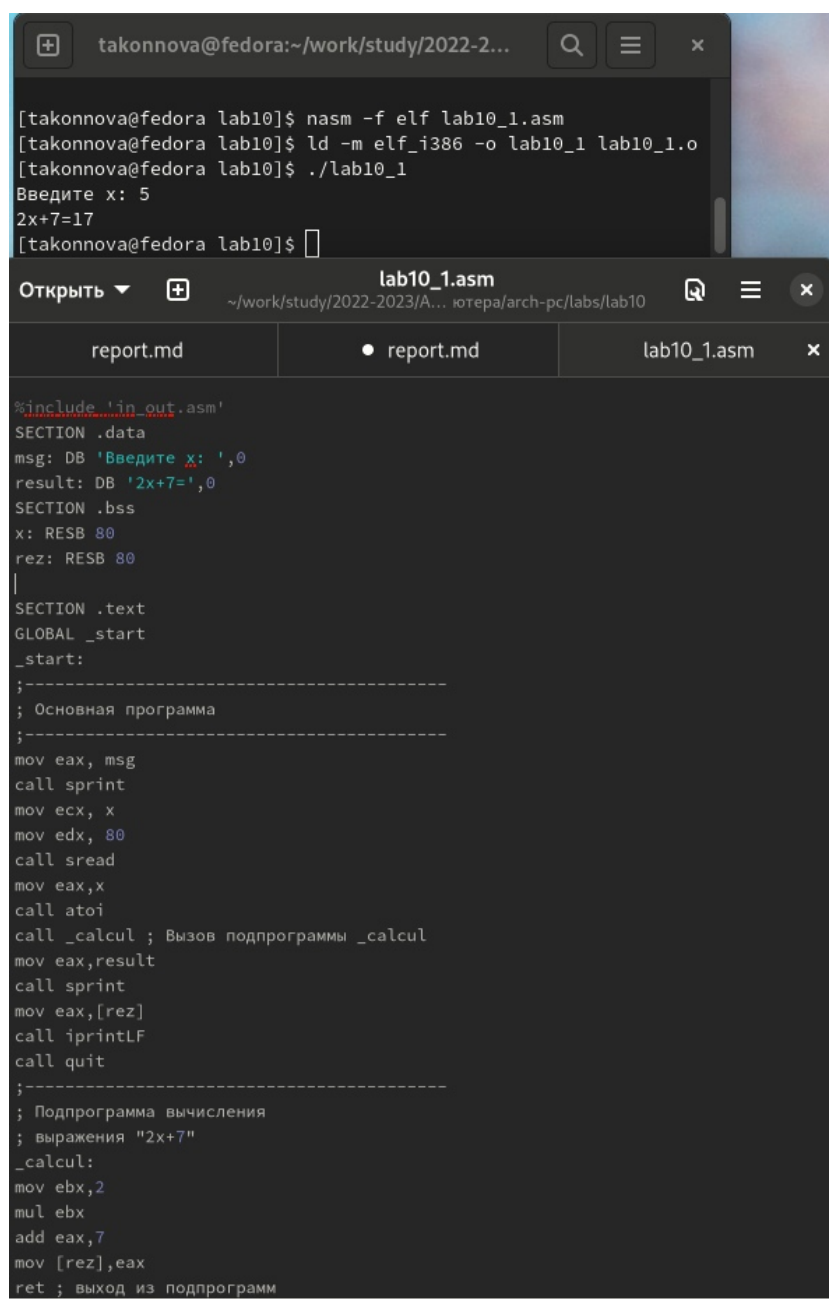
1. Создаем каталог для выполнения лабораторной работы No 10, переходим в него и создаем файл lab10-1.asm:

```
mkdir ~/work/arch-pc/lab10
```

```
cd ~/work/arch-pc/lab10
```

```
touch 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 изучаем текст программы:(рис. 1.1)



```
[takonnova@fedora lab10]$ nasm -f elf lab10_1.asm
[takonnova@fedora lab10]$ ld -m elf_i386 -o lab10_1 lab10_1.o
[takonnova@fedora lab10]$ ./lab10_1
Введите x: 5
2x+7=17
[takonnova@fedora lab10]$
```

```
%include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2x+7=',0
SECTION .bss
x: RESB 80
rez: RESB 80
|
SECTION .text
GLOBAL _start
_start:
;-----
; Основная программа
;-----
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax, x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax, result
call sprint
mov eax, [rez]
call iprintLF
call quit
;-----
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx, 2
mul ebx
add eax, 7
mov [rez], eax
ret ; выход из подпрограмм
```

Рис. 1.1: lab10\_1.asm

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

После следующей инструкции call \_calcul, которая передает управление подпрограмме \_calcul, будут выполнены инструкции подпрограммы:

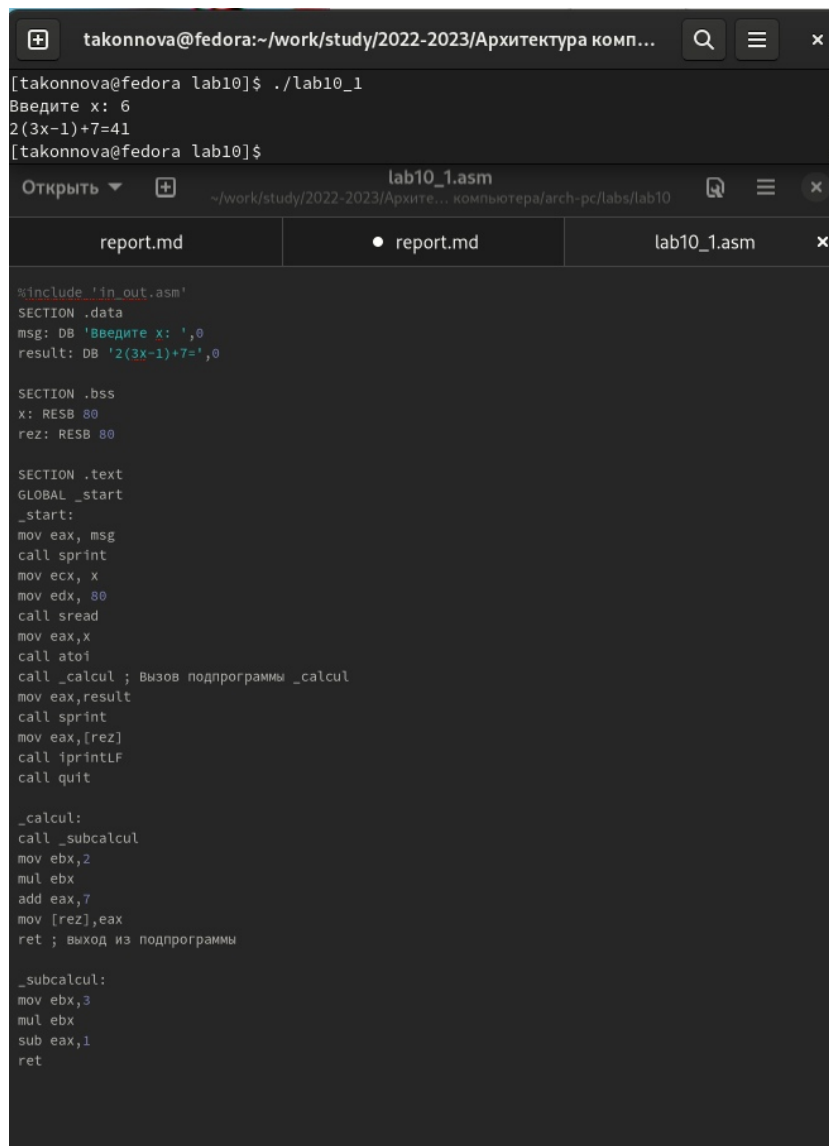
```
mov ebx,2
mul ebx
add eax,7
mov [rez,eax
ret
```

Инструкция `ret` является последней в подпрограмме и ее исполнение приводит к возвращению в основную программу к инструкции, следующей за инструкцией `call`, которая вызвала данную подпрограмму.

Последние строки программы реализуют вывод сообщения (`call sprint`), результата вычисления (`call iprintLF`) и завершение программы (`call quit`).

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

- 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.  $x$  передается в подпрограмму `_calcul` из нее 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. (рис. 1.2)



```
[takonnova@fedora lab10]$ ./lab10_1
Введите x: 6
2(3x-1)+7=41
[takonnova@fedora lab10]$
```

```
lab10_1.asm
~/work/study/2022-2023/Архите... компьютера/arch-pc/labs/lab10

report.md • report.md lab10_1.asm x

#include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2(3x-1)+7=',0

SECTION .bss
x: RESB 80
rez: RESB 80

SECTION .text
GLOBAL _start
_start:
mov eax, msg
call sprint
mov ecx, x
mov edx, 80
call sread
mov eax, x
call atoi
call _calcul ; Вызов подпрограммы _calcul
mov eax, result
call sprint
mov eax, [rez]
call iprintLF
call quit

_calcul:
call _subcalcul
mov ebx, 2
mul ebx
add eax, 7
mov [rez], eax
ret ; выход из подпрограммы

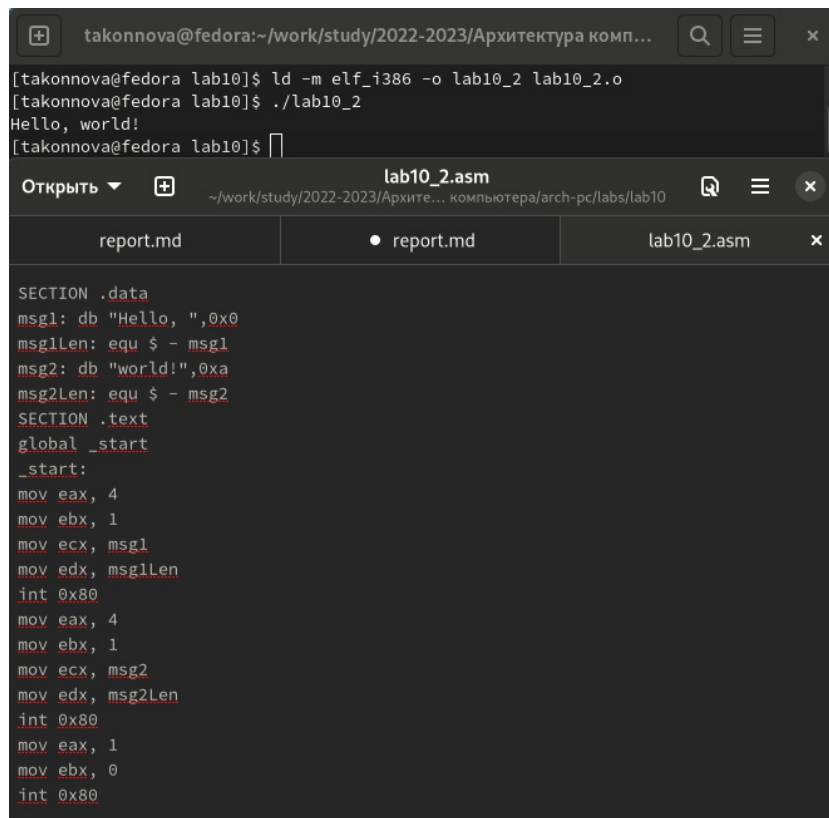
_subcalcul:
mov ebx, 3
mul ebx
sub eax, 1
ret
```

Рис. 1.2: subcalcul

### 1.3 9.3.2. Отладка программ с помощью GDB

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

(рис. 1.3)



The image shows a terminal window and a code editor. The terminal window at the top shows the following commands and output:

```
[takonnova@fedora lab10]$ ld -m elf_i386 -o lab10_2 lab10_2.o
[takonnova@fedora lab10]$ ./lab10_2
Hello, world!
[takonnova@fedora lab10]$
```

The code editor below shows the contents of the file `lab10_2.asm`:

```
SECTION .data
msg1: db "Hello, ",0x0
msg1Len: equ $ - msg1
msg2: db "world!",0xa
msg2Len: equ $ - msg2
SECTION .text
global _start
_start:
mov eax, 4
mov ebx, 1
mov ecx, msg1
mov edx, msg1Len
int 0x80
mov eax, 4
mov ebx, 1
mov ecx, msg2
mov edx, msg2Len
int 0x80
mov eax, 1
mov ebx, 0
int 0x80
```

Рис. 1.3: 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’.

```
nasm -f elf -g -l lab10-2.lst lab10-2.asm
```

```
ld -m elf_i386 -o lab10-2 lab10-2.o
```

Загрузите исполняемый файл в отладчик gdb:

(рис. 1.4)



```

[takonnova@fedora lab10]$ nasm -f elf -g lab10_2.asm
[takonnova@fedora lab10]$ ld -m elf_i386 -o lab10_2 lab10_2.o
[takonnova@fedora lab10]$ gdb lab10_2
GNU gdb (GDB) Fedora 11.2-3.fc36
Copyright (C) 2022 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 lab10_2...
(gdb) run
Starting program: /home/takonnova/work/study/2022-2023/Архитектура компьютера/arch-pc/labs/lab10/lab10_2

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.
Downloading separate debug info for /home/takonnova/work/study/2022-2023/Архитектура компьютера/arch-pc/lab
run
Hello, world!
[Inferior 1 (process 35083) exited normally]
(gdb)
(gdb)
(gdb) run
Starting program: /home/takonnova/work/study/2022-2023/Архитектура компьютера/arch-pc/labs/lab10/lab10_2
Hello, world!
[Inferior 1 (process 35096) exited normally]
(gdb)

```

Рис. 1.4: lab10-2.asm

`gdb lab10-2`

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

`(gdb) run`

Starting program: `~/work/arch-pc/lab10/lab10-2`

Hello, world!

Выводится Inferior 1 (process 10220) exited normally

`(gdb)`

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

```
(gdb) break _start
```

(рис. 1.5)

```
takonnova@fedora:~/work/study/2022-2023/Архитектура ком...
ch-pc/labs/lab10/lab10_2
Hello, world!
[Inferior 1 (process 35096) exited normally]
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab10_2.asm, line 9.
(gdb) run
Starting program: /home/takonnova/work/study/2022-2023/Архитектура компьютера/ar
ch-pc/labs/lab10/lab10_2

Breakpoint 1, _start () at lab10_2.asm:9
9      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) 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. 1.5: breakpoint

Breakpoint 1 at 0x8049000: file lab10-2.asm, line 12.

(gdb) run

Starting program: ~/work/arch-pc/lab10/lab10-2

Breakpoint 1, \_start () at lab10-2.asm:12

12 mov eax, 4

Посмотрим дисассимилированный код программы с помощью команды `disassemble` начиная с метки `_start`

(gdb) disassemble \_start

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

(gdb) set disassembly-flavor intel

(gdb) disassemble \_start

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

Включим режим псевдографики для более удобного анализа программы (рис. 10.2):

(gdb) layout asm

(gdb) layout regs

В этом режиме есть три окна после выполнения команды `si`, (показано до нее):

- В верхней части видны названия регистров и их текущие значения;
- В средней части виден результат дисассимилирования программы;
- Нижняя часть доступна для ввода команд.

(рис. 1.6)

The screenshot shows a terminal window with a dark background. At the top, the window title is 'takonnova@fedora:~/work/study/2022-2023/Архитектура ком...'. Below the title bar, there is a large black rectangle with the text '[ Register Values Unavailable ]'. Below this, a list of assembly instructions is displayed, each preceded by an address and a label. The instructions are: '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', and '0x804902a <\_start+42> int 0x80'. At the bottom of the terminal, the text 'native process 36058 In: \_start' is followed by 'L9' and 'PC: 0x8049000'. Below this, the command '(gdb) layout regs' is entered, followed by a prompt '(gdb) '.

```
takonnova@fedora:~/work/study/2022-2023/Архитектура ком...
[ Register Values Unavailable ]

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 36058 In: _start L9 PC: 0x8049000
(gdb) layout regs
(gdb) 
```

Рис. 1.6: layout

## 1.4 10.4.2.1. 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`)(ввела эту команду, она не сохранилась на экране, к сожалению):

```
(gdb) info breakpoints
```

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции (см. рис. 10.3). Определим адрес предпоследней инструкции (`mov ebx,0x0`) и установим точку останова.

```
(gdb) break *
```

Посмотрим информацию о всех установленных точках останова:

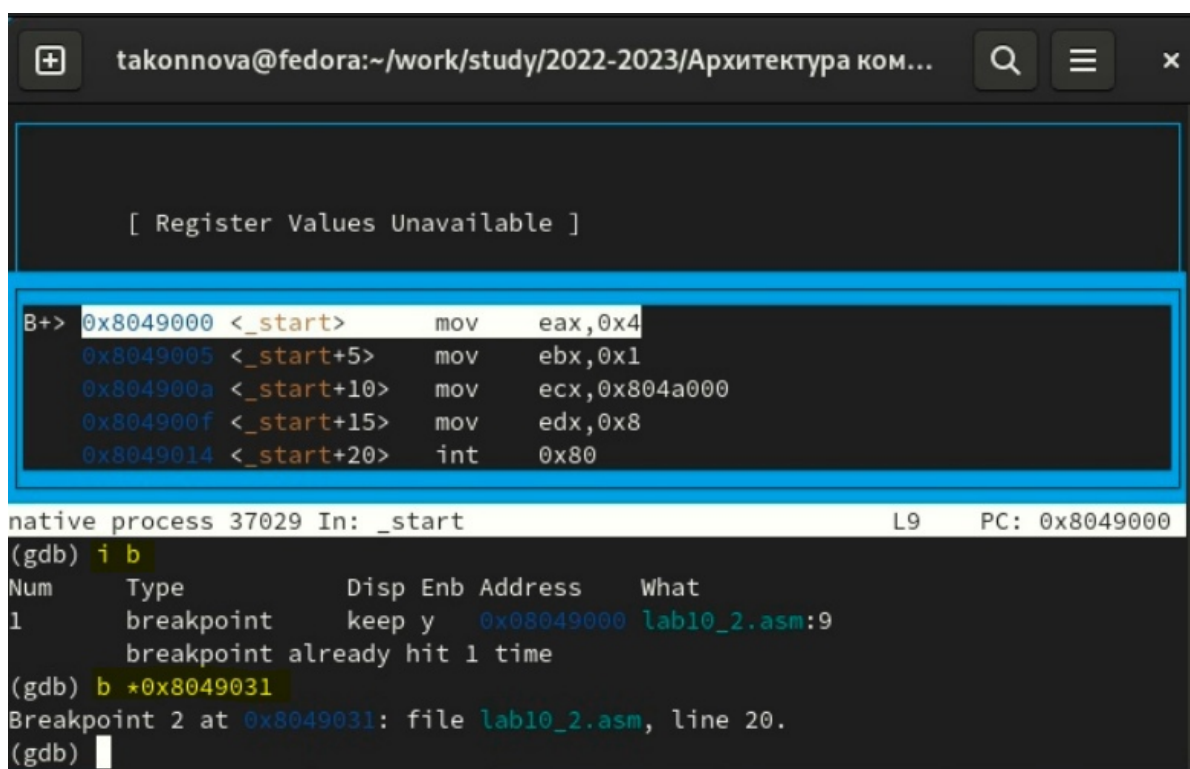
```
(gdb) i b
```

(рис. 1.7) (рис. 1.8)

```
takonnova@fedora:~/work/study/2022-2023/Архитектура ком...

(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. 1.7: intel change



The screenshot shows a GDB terminal window with the title bar "takonnova@fedora:~/work/study/2022-2023/Архитектура ком...". The main display area shows assembly code for a native process 37029, starting at address 0x8049000. The code consists of five instructions: `mov eax, 0x4`, `mov ebx, 0x1`, `mov ecx, 0x804a000`, `mov edx, 0x8`, and `int 0x80`. Below the assembly code, the status bar indicates "native process 37029 In: \_start" and "PC: 0x8049000". The command prompt shows the user has entered `(gdb) i b`, and the output displays a table of breakpoints. The first breakpoint is at address 0x8049000, labeled "lab10\_2.asm:9", and has been hit 1 time. The second breakpoint is at address 0x8049031, labeled "lab10\_2.asm, line 20". The command prompt also shows `(gdb) b *0x8049031`.

```
[ Register Values Unavailable ]

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

native process 37029 In: _start                                L9      PC: 0x8049000
(gdb) i b
Num      Type          Disp Enb Address      What
1        breakpoint    keep y  0x08049000 lab10_2.asm:9
          breakpoint already hit 1 time
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab10_2.asm, line 20.
(gdb)
```

Рис. 1.8: intel change

## 1.5 10.4.2.2. 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. Значения каких регистров изменяются?

Ответ: меняются одномерно значения регистров, а именно `eax`, `ecx`, `edx`, `ebx`

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

`(gdb) info registers`

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/1sb &msg1` также можно посмотреть содержимое переменной. Посмотрим значение переменной `msg1` по имени

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

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`

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}msg1='h'
```

```
(gdb) x/1sb &msg1
```

```
0x804a000 : "hello,"
```

```
(gdb)
```

(рис. 1.9)

```

0x804a000 <msg1>:      "hello, "
(gdb) set {char}&msg1='h'
(gdb) set {char}0x804a001='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hhlllo, "
(gdb) set {char}0x804a008='L'
(gdb) set {char}0x804a00b=' '
0x804a008 <msg2>:      "Lor d!\n"
(gdb)

```

Рис. 1.9: intel change

Заменяем любой символ во второй переменной msg2. Чтобы посмотреть значения регистров используется команда print /F “val” (перед именем регистра обязательно ставится префикс \$): p/F \$

Выведем в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра edx. С помощью команды set изменим значение регистра ebx:

```

(gdb) set $ebx='2'
(gdb) p/s $ebx
$3 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$4 = 2
(gdb)

```

(рис. 1.10)

```

$5 = 8
(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

```

Рис. 1.10: change

(рис. 1.11)

```

$2 = 100
(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

```

Рис. 1.11: change

Объясните разницу вывода команд p/s \$ebx.

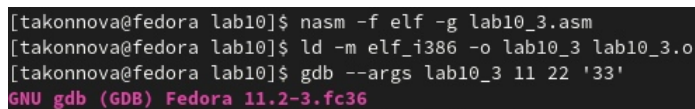
- Ответ: выводились в разном виде, то есть подразумевает вывод либо номера числа в таблице ASCII, либо само значение числа. Завершила выполнение программы с помощью команды quit. На экране не отобразилось.

## 1.6 10.4.2.3. Обработка аргументов командной строки в GDB

Скопируем файл lab9-2.asm, созданный при выполнении лабораторной работы No9, с программой выводящей на экран аргументы командной строки (Листинг 9.2) в файл с именем lab10-3.asm:

```
ср ~/work/arch-pc/lab09/lab9-2.asm ~/work/arch-pc/lab10/lab10-3.asm
```

Создаем исполняемый файл. (рис. 1.12)



```
[takonnova@fedora lab10]$ nasm -f elf -g lab10_3.asm
[takonnova@fedora lab10]$ ld -m elf_i386 -o lab10_3 lab10_3.o
[takonnova@fedora lab10]$ gdb --args lab10_3 11 22 '33'
GNU gdb (GDB) Fedora 11.2-3.fc36
```

Рис. 1.12: лаб10\_3

```
nasm -f elf -g -l lab10-3.lst lab10-3.asm
```

```
ld -m elf_i386 -o lab10-3 lab10-3.o
```

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

```
gdb -args lab10-3 аргумент1 аргумент 2 'аргумент 3'
```

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 ее.

```
(gdb) b _start
```

```
(gdb) run
```

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
```

```
0xffffd200: 0x05
```

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] храниться адрес первого аргумента, по адресу esp+12 – второго и т.д. (рис. 1.12)

```

takonnova@fedora:~/work/study/2022-2023/Архитектура ком...
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab10_3...
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab10_3.asm, line 5.
(gdb) run
Starting program: /home/takonnova/work/study/2022-2023/Архитектура компьютера/arch-pc/labs/lab10/lab10_3 11 22 33

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) T
Please answer y or [n].

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

Breakpoint 1, _start () at lab10_3.asm:5
5      pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb) x/x $esp
0xffffd128: 0x00000004
(gdb)

```

Рис. 1.13: gdb

```

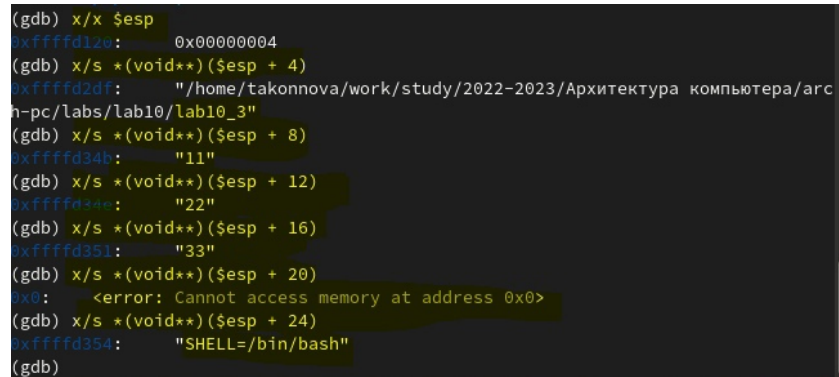
(gdb) x/s *(void**)($esp + 4)
0xffffd358: "~/lab10-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd3bc: "аргумент1"
(gdb) x/s *(void**)($esp + 12)
0xffffd3ce: "аргумент"
(gdb) x/s *(void**)($esp + 16)
0xffffd3df: "2"
(gdb) x/s *(void**)($esp + 20)
0xffffd3e1: "аргумент 3"

```

(gdb) x/s \*(void\*\*)(esp + 24)

0x0: error: Cannot access memory at address 0x0

(gdb)



```
(gdb) x/s $esp
0xffffd12e: 0x00000004
(gdb) x/s *(void**)(esp + 4)
0xffffd2df: "/home/takonnova/work/study/2022-2023/Архитектура компьютера/arc
n-pc/labs/lab10/lab10_3"
(gdb) x/s *(void**)(esp + 8)
0xffffd34b: "11"
(gdb) x/s *(void**)(esp + 12)
0xffffd38e: "22"
(gdb) x/s *(void**)(esp + 16)
0xffffd3d1: "33"
(gdb) x/s *(void**)(esp + 20)
0x0: <error: Cannot access memory at address 0x0>
(gdb) x/s *(void**)(esp + 24)
0xffffd354: "SHELL=/bin/bash"
(gdb)
```

Рис. 1.14: gdb

Объясним, почему шаг изменения адреса равен 4 ([esp+4, [esp+8, [esp+12 и т.д.)

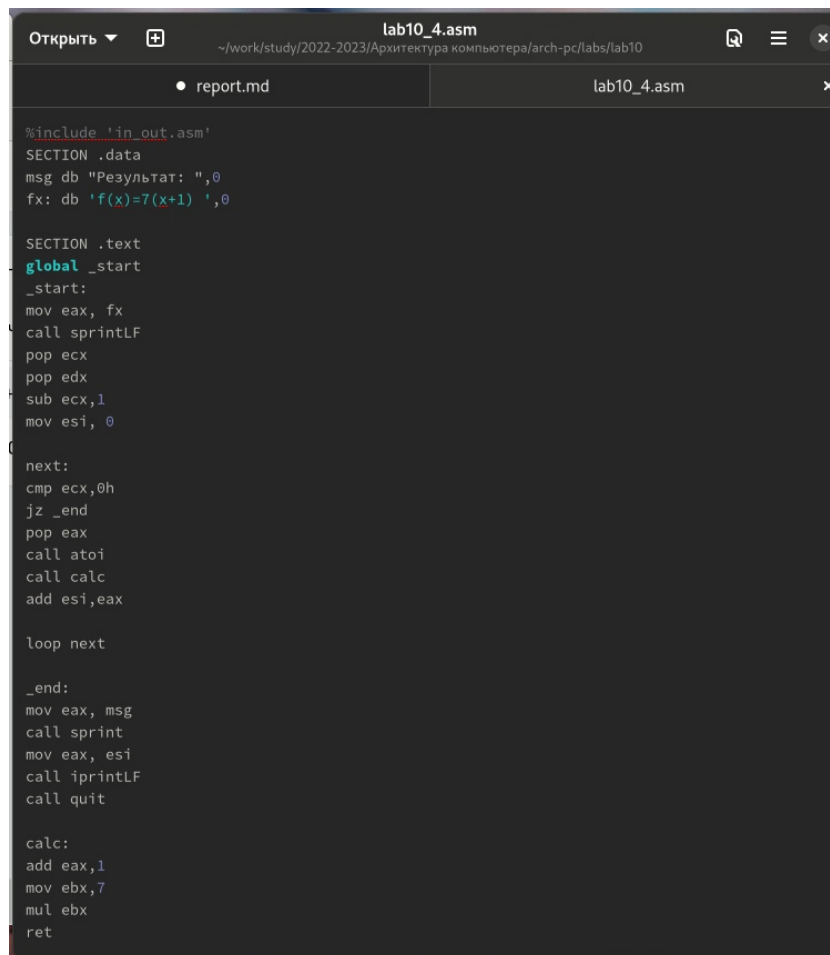
- Ответ:

Так как число аргументов = 5 – это имя программы lab10\_3 и аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’. В других позициях по адресу [esp + 4] располагается адрес в памяти, там и находится имя пр-ммы, по адресу [esp + 8] хранится адрес первого аргумента, по адресу [esp + 12] второго.

## 2 10.5. Самостоятельная работа

1. Преобразуйте программу из лабораторной работы No9 (Задание No1 для самостоятельной работы), реализовав вычисление значения функции  $f(x)$  как подпрограмму.

(рис. 2.1)



```
lab10_4.asm
~/work/study/2022-2023/Архитектура компьютера/arch-pc/labs/lab10

report.md lab10_4.asm

%include 'in_out.asm'
SECTION .data
msg db "Результат: ",0
fx: db 'f(x)=7(x+1) ',0

SECTION .text
global _start
_start:
mov eax, fx
call sprintLF
pop ecx
pop edx
sub ecx,1
mov esi, 0

next:
cmp ecx,0h
jz _end
pop eax
call atoi
call calc
add esi,eax

loop next

_end:
mov eax, msg
call sprint
mov eax, esi
call iprintLF
call quit

calc:
add eax,1
mov ebx,7
mul ebx
ret
```

Рис. 2.1: лаб10\_5

```

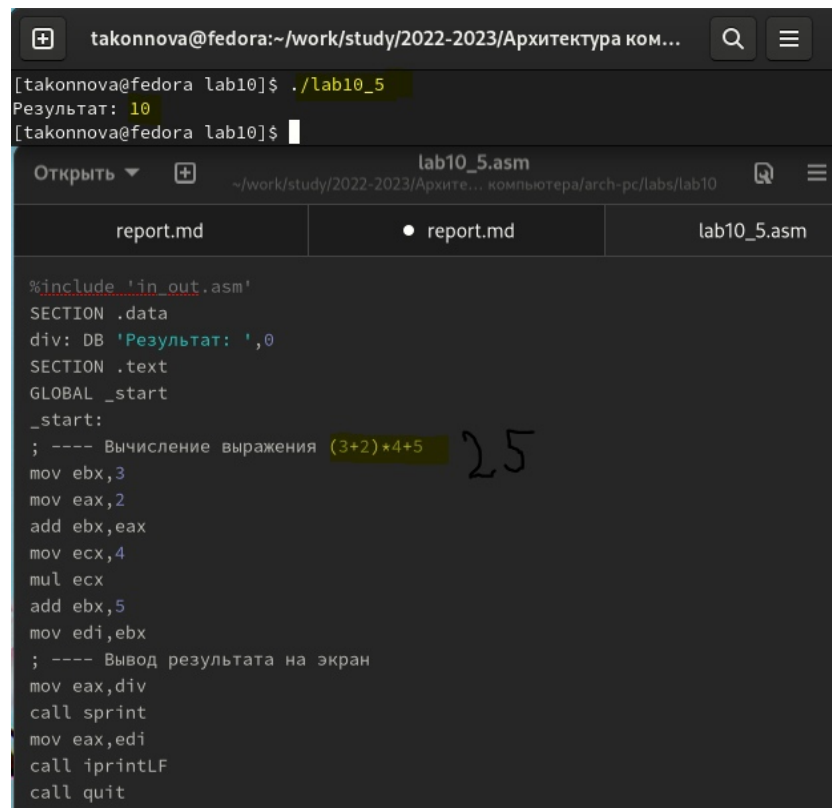
[takonnova@fedora lab10]$ ./lab10_4
f(x)=7(x+1)
Результат: 0
[takonnova@fedora lab10]$ ./lab10_4 11 22 33 44 55 66
f(x)=7(x+1)
Результат: 1659
[takonnova@fedora lab10]$

```

Рис. 2.2: gdb

2. В листинге 10.3 приведена программа вычисления выражения  $(3 + 2) * 4 + 5$ . При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

(рис. 2.3)



```

takonnova@fedora:~/work/study/2022-2023/Архитектура ком...
[takonnova@fedora lab10]$ ./lab10_5
Результат: 10
[takonnova@fedora lab10]$

lab10_5.asm
~/work/study/2022-2023/Архите... компьютера/arch-pc/labs/lab10
report.md • report.md lab10_5.asm

%include 'in_out.asm'
SECTION .data
div: DB 'Результат: ',0
SECTION .text
GLOBAL _start
_start:
; ---- Вычисление выражения (3+2)*4+5
mov ebx,3
mov eax,2
add ebx,eax
mov ecx,4
mul ecx
add ebx,5
mov edi,ebx
; ---- Вывод результата на экран
mov eax,div
call sprint
mov eax,edi
call iprintLF
call quit

```

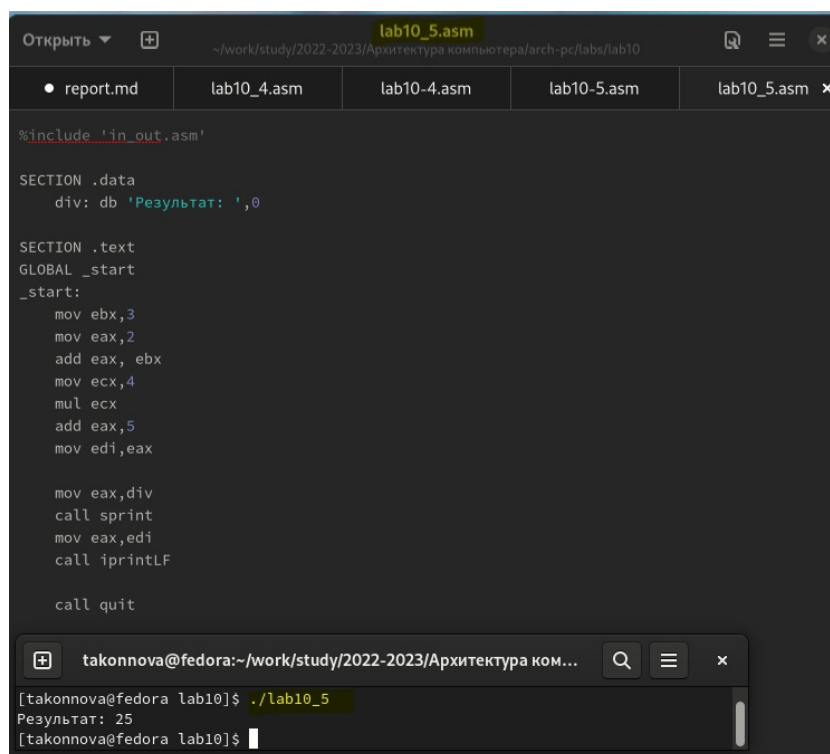
Рис. 2.3: лаб10\_5

Да, ответ неверен. Здесь регистр ecx умножают на исходное значение регистра eax, а не на значение, полученное после сложения eax и ebx. И результат сложения



сохраняется в `ebx`, в то время как `ecx` умножается на `eax`, он = 2. Затем по логике нужно добавлять 5 к `eax` (20, а не 2), 5 суммируется с `ebx`,=5. Получается 8 и 10, а не 20 и 25. Еще в `edi` необходимо записывать значение `eax`, а не `ebx`.

Исправим ошибки и получим верный ответ. (рис. 2.4)



```
Открыть ▾ + lab10_5.asm
~/work/study/2022-2023/Архитектура компьютера/arch-pc/labs/lab10
• report.md lab10_4.asm lab10-4.asm lab10-5.asm lab10_5.asm x

%include 'in_out.asm'

SECTION .data
div: db 'Результат: ',0

SECTION .text
GLOBAL _start
_start:
    mov ebx,3
    mov eax,2
    add eax, ebx
    mov ecx,4
    mul ecx
    add eax,5
    mov edi,eax

    mov eax,div
    call sprint
    mov eax,edi
    call iprintLF

    call quit

takonnova@fedora:~/work/study/2022-2023/Архитектура ком... 🔍 ☰ x
[takonnova@fedora lab10]$ ./lab10_5
Результат: 25
[takonnova@fedora lab10]$
```

Рис. 2.4: Правки в работе

## **3 Выводы**

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