

# **Отчет по лабораторной работе №9**

**Дисциплина: архитектура компьютера**

Лаптев Тимофей Сергеевич

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>8</b>
4.1	Релаксация подпрограмм в NASM . . . . .	8
4.1.1	Отладка программ с помощью GDB . . . . .	11
4.1.2	Добавление точек останова . . . . .	15
4.1.3	Работа с данными программы в GDB . . . . .	16
4.1.4	Обработка аргументов командной строки в GDB . . . . .	18
4.2	Задание для самостоятельной работы . . . . .	19
<b>5</b>	<b>Выводы</b>	<b>25</b>
<b>6</b>	<b>Список литературы</b>	<b>26</b>

# Список иллюстраций

4.1	Создание рабочего каталога . . . . .	8
4.2	Запуск программы из листинга . . . . .	9
4.3	Изменение программы первого листинга . . . . .	9
4.4	Запуск программы в отладчике . . . . .	12
4.5	Проверка программы отладчиком . . . . .	12
4.6	Запуск отладчика с брейкпойнтом . . . . .	13
4.7	Дисассимилирование программы . . . . .	14
4.8	Режим псевдографики . . . . .	14
4.9	Список брейкпойнтов . . . . .	15
4.10	Добавление второй точки останова . . . . .	15
4.11	Просмотр содержимого регистров . . . . .	16
4.12	Просмотр содержимого переменных двумя способами . . . . .	16
4.13	Изменение содержимого переменных двумя способами . . . . .	17
4.14	Просмотр значения регистра разными представлениями . . . . .	17
4.15	Примеры использования команды set . . . . .	18
4.16	Подготовка новой программы . . . . .	18
4.17	Проверка работы стека . . . . .	19
4.18	Измененная программа предыдущей лабораторной работы . . . . .	20
4.19	Поиск ошибки в программе через пошаговую отладку . . . . .	22
4.20	Проверка корректировок в программе . . . . .	23

## **Список таблиц**

# 1 Цель работы

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

## 2 Задание

1. Реализация подпрограмм в NASM
2. Отладка программ с помощью GDB
3. Самостоятельное выполнение заданий по материалам лабораторной работы

### 3 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.

Можно выделить следующие типы ошибок:

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

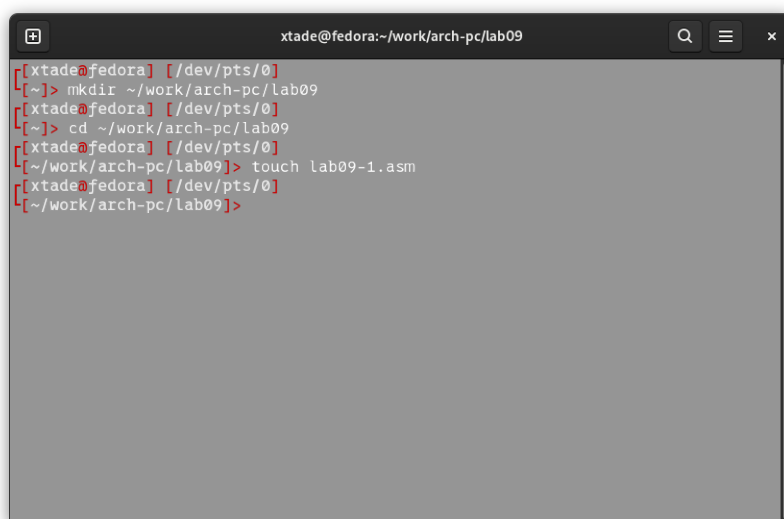
Второй этап — поиск местонахождения ошибки. Некоторые ошибки обнаружить довольно трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга.

Третий этап — выяснение причины ошибки. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы. Последний этап — исправление ошибки. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново.

## 4 Выполнение лабораторной работы

### 4.1 Релазация подпрограмм в NASM

Создаю каталог для выполнения лабораторной работы №9 (рис. -fig. 4.1).

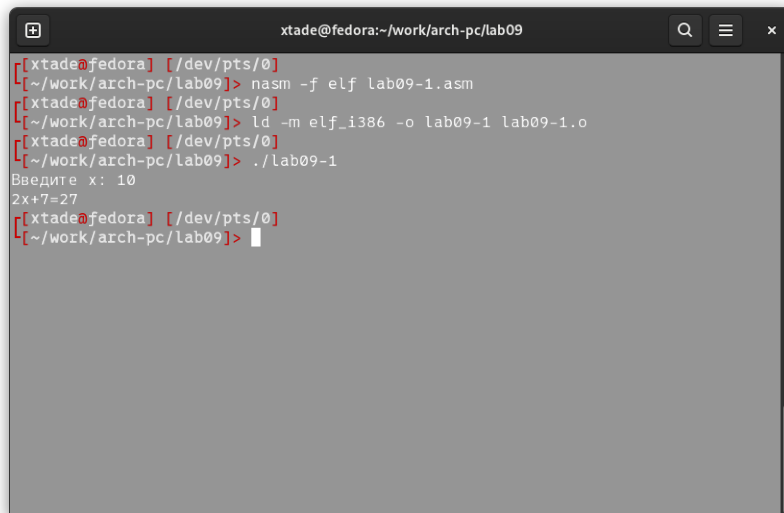


```
xtade@fedora: ~/work/arch-pc/lab09
[xtade@fedora] [/dev/pts/0]
[~]> mkdir ~/work/arch-pc/lab09
[xtade@fedora] [/dev/pts/0]
[~]> cd ~/work/arch-pc/lab09
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> touch lab09-1.asm
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]>
```

Рис. 4.1: Создание рабочего каталога

Копирую в файл код из листинга, компилирую и запускаю его, данная программа выполняет вычисление функции (рис. -fig. 4.2).

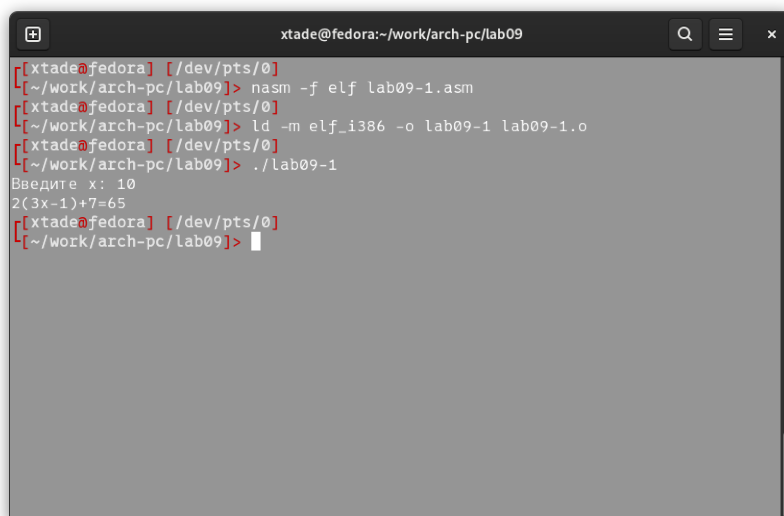




```
xtade@fedora:~/work/arch-pc/lab09
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> nasm -f elf lab09-1.asm
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> ld -m elf_i386 -o lab09-1 lab09-1.o
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> ./lab09-1
Введите x: 10
2x+7=27
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]>
```

Рис. 4.2: Запуск программы из листинга

Изменяю текст программы, добавив в нее подпрограмму, теперь она вычисляет значение функции для выражения  $f(g(x))$  (рис. -fig. 4.3).



```
xtade@fedora:~/work/arch-pc/lab09
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> nasm -f elf lab09-1.asm
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> ld -m elf_i386 -o lab09-1 lab09-1.o
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]> ./lab09-1
Введите x: 10
2(3x-1)+7=65
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09]>
```

Рис. 4.3: Изменение программы первого листинга

Код программы:

```
%include 'in_out.asm'
```

## SECTION .data

```
msg: DB 'Введите x: ', 0
result: DB '2(3x-1)+7=', 0
```

## SECTION .bss

```
x: RESB 80
res: 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
```

```
mov eax, result
```

```
call sprint
```

```
mov eax, [res]
```

```
call iprintLF
```

```
call quit
```

```

_calcul:
push eax
call _subcalcul

mov ebx, 2
mul ebx
add eax, 7

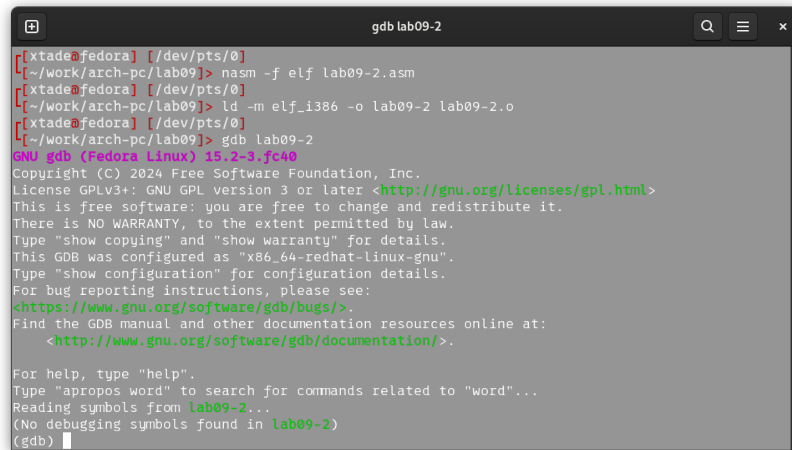
mov [res], eax
pop eax
ret

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

```

#### 4.1.1 Отладка программ с помощью GDB

В созданный файл копирую программу второго листинга, транслирую с созданием файла листинга и отладки, компону и запускаю в отладчике (рис. -fig. 4.4).

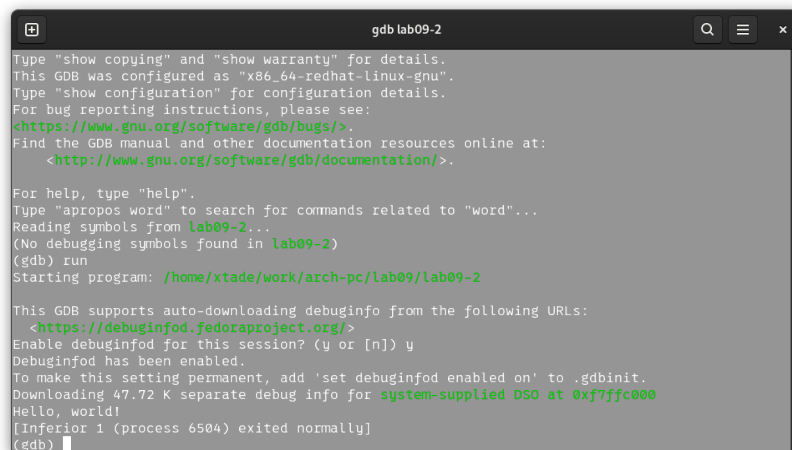


```
[xtade@fedora] [/dev/pts/0]
[~/work/arch-pc/lab09] nasm -f elf lab09-2.asm
[~/work/arch-pc/lab09] ld -m elf_i386 -o lab09-2 lab09-2.o
[~/work/arch-pc/lab09] gdb lab09-2
GNU gdb (Fedora Linux) 15.2-3.fc40
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <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 lab09-2...
(No debugging symbols found in lab09-2)
(gdb)
```

Рис. 4.4: Запуск программы в отладчике

Запустив программу командой `run`, я убедился в том, что она работает исправно (рис. -fig. 4.5).



```
(gdb) run
Starting program: /home/xtade/work/arch-pc/lab09/lab09-2

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 47.72 K separate debug info for system-supplied DSO at 0xf7ffc000
Hello, world!
[Inferior 1 (process 6504) exited normally]
(gdb)
```

Рис. 4.5: Проверка программы отладчиком

Для более подробного анализа программы добавляю брейкпоинт на метку `_start` и снова запускаю отладку (рис. -fig. 4.6).

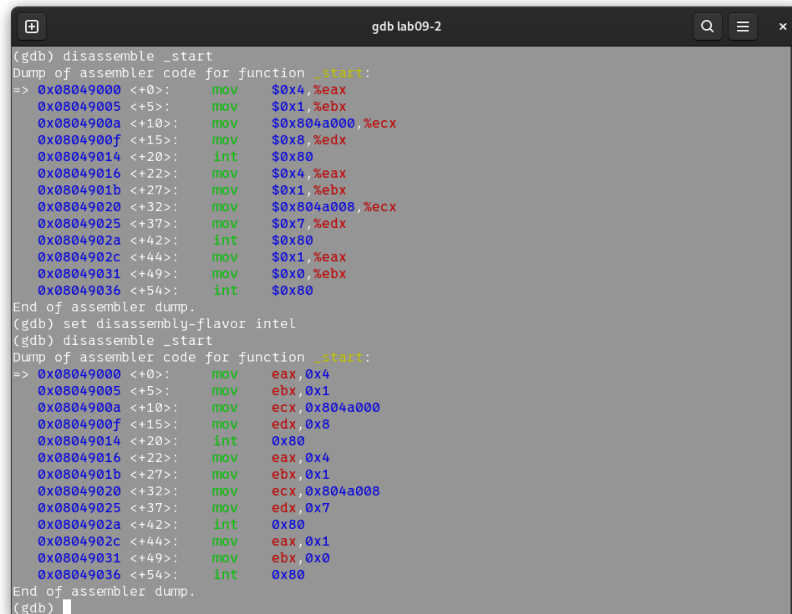


```
(gdb) break _start
Breakpoint 1 at 0x0049000
(gdb) run
Starting program: /home/xtade/work/arch-pc/lab09/lab09-2
Breakpoint 1, 0x0049000 in _start ()
(gdb)
```

Рис. 4.6: Запуск отладчика с брейкпоинтом

Далее смотрю дисассимилированный код программы, перевожу на команд с синтаксисом Intel *amd топчик* (рис. -fig. 4.7).

Различия между синтаксисом АТТ и Intel заключаются в порядке операндов (АТТ - Операнд источника указан первым. Intel - Операнд назначения указан первым), их размере (АТТ - размер операндов указывается явно с помощью суффиксов, непосредственные операнды предваряются символом \$; Intel - Размер операндов неявно определяется контекстом, как ax, eax, непосредственные операнды пишутся напрямую), именах регистров(АТТ - имена регистров предваряются символом %, Intel - имена регистров пишутся без префиксов).

A screenshot of a GDB window titled 'gdb lab09-2'. The user has entered the command '(gdb) disassemble \_start'. The output shows a dump of assembly code for the function '\_start'. The code is listed with addresses from 0x08049000 to 0x08049036 and corresponding assembly instructions like 'mov \$0x4,%eax', 'mov \$0x1,%ebx', etc. The user then sets the disassembly flavor to 'intel' and runs 'disassemble \_start' again, showing the same code but with Intel syntax. The window has a search bar and standard window controls at the top.

```
(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. 4.7: Дисассимилирование программы

Включаю режим псевдографики для более удобного анализа программы (рис. -fig. 4.8).

A screenshot of a GDB window titled 'gdb lab09-2'. The 'Register group: general' window is open, showing the values of registers: eax (0x0), ecx (0x0), edx (0x0), ebx (0x0), esp (0xffffcb0), and ebp (0x0). Below the register window, the assembly code for the '\_start' function is displayed, with the first few instructions highlighted in blue. The GDB console at the bottom shows the status of the native process 7650 (asm) in the '\_start' function, with the PC at 0x8049000. The user has entered the command '(gdb) regs' and the output shows the register values. The window has a search bar and standard window controls at the top.

```
Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffcb0 0xffffcb0
ebp      0x0      0x0

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

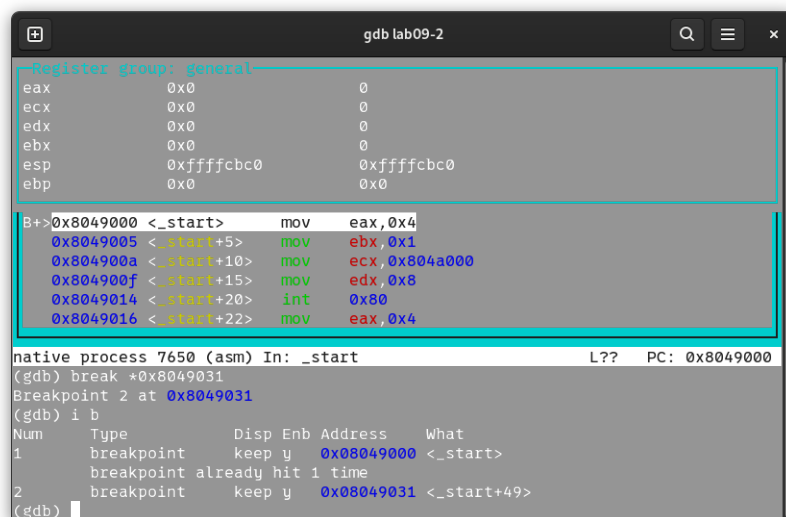
native process 7650 (asm) In: _start L?? PC: 0x8049000
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.

Breakpoint 1, 0x08049000 in _start ()
(gdb) regs
Undefined command: "regs". Try "help".
(gdb)
```

Рис. 4.8: Режим псевдографики

### 4.1.2 Добавление точек останова

Проверяю в режиме псевдографики, что брейкпоинт сохранился (рис. -fig. 4.9).



The screenshot shows the GDB interface with the following content:

```
gdb lab09-2

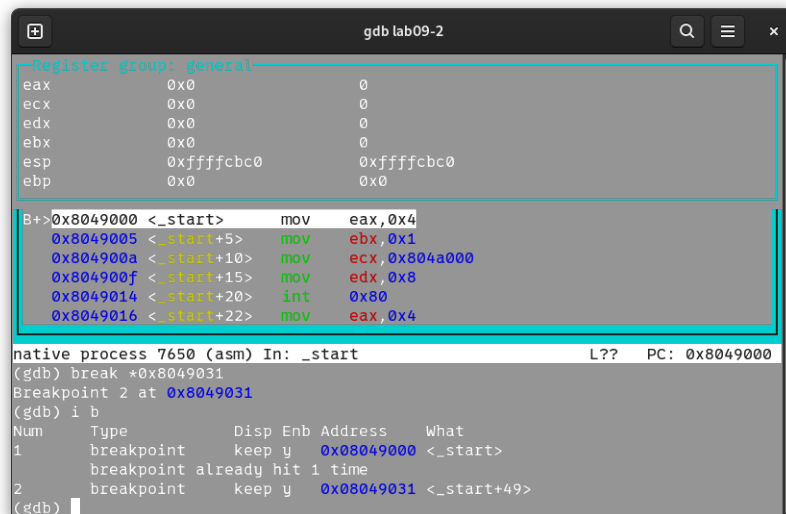
--Register group: general--
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xfffffcb0 0xfffffcb0
ebp      0x0      0x0

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

native process 7650 (asm) In: _start      L??  PC: 0x8049000
(gdb) break *0x8049031
Breakpoint 2 at 0x8049031
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y  0x08049000 <_start>
         breakpoint already hit 1 time
2        breakpoint      keep y  0x08049031 <_start+49>
(gdb)
```

Рис. 4.9: Список брейкпоинтов

Устанавливаю еще одну точку останова по адресу инструкции (рис. -fig. 4.10).



The screenshot shows the GDB interface with the following content:

```
gdb lab09-2

--Register group: general--
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xfffffcb0 0xfffffcb0
ebp      0x0      0x0

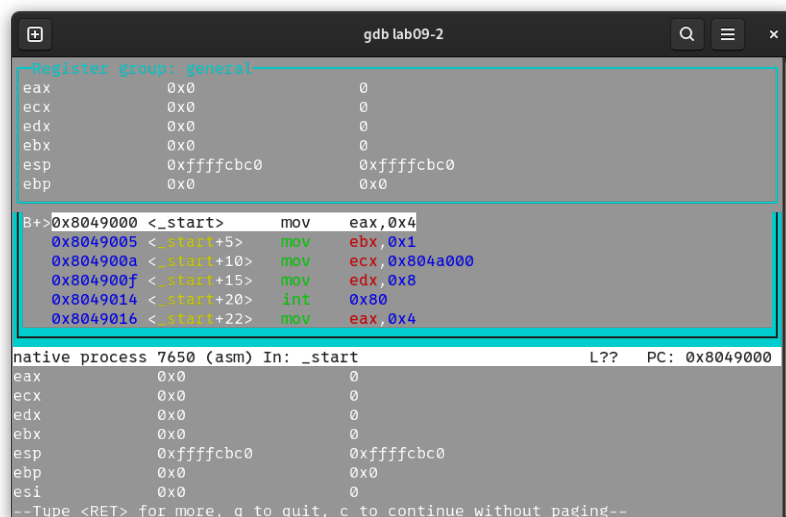
B+>0x8049000 <_start>    mov     eax,0x4
0x8049005 <_start+5>    mov     ebx,0x1
0x804900a <_start+10>   mov     ecx,0x804a000
0x804900f <_start+15>   mov     edx,0x8
0x8049014 <_start+20>   int     0x80
0x8049016 <_start+22>   mov     eax,0x4

native process 7650 (asm) In: _start      L??  PC: 0x8049000
(gdb) break *0x8049031
Breakpoint 2 at 0x8049031
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y  0x08049000 <_start>
         breakpoint already hit 1 time
2        breakpoint      keep y  0x08049031 <_start+49>
(gdb)
```

Рис. 4.10: Добавление второй точки останова

### 4.1.3 Работа с данными программы в GDB

Просматриваю содержимое регистров командой `info registers` (рис. -fig. 4.11).



The screenshot shows the GDB interface with the command `info registers` executed. The output displays the values of the general-purpose registers: `eax`, `ecx`, `edx`, `ebx`, `esp`, and `ebp`. Below the register list, the assembly code for the current instruction is shown, along with the values of the registers used in the instructions.

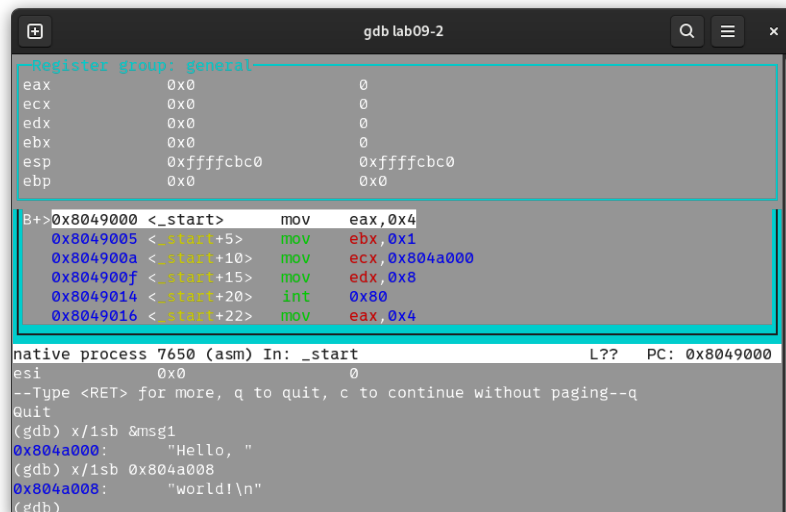
```
gdb lab09-2
~Register group: general~
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffcb0 0xffffcb0
ebp      0x0      0x0

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

native process 7650 (asm) In: _start L?? PC: 0x8049000
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffcb0 0xffffcb0
ebp      0x0      0
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
```

Рис. 4.11: Просмотр содержимого регистров

Смотрю содержимое переменных по имени и по адресу (рис. -fig. 4.12).



The screenshot shows the GDB interface with the command `info registers` executed. The output displays the values of the general-purpose registers: `eax`, `ecx`, `edx`, `ebx`, `esp`, and `ebp`. Below the register list, the assembly code for the current instruction is shown, along with the values of the registers used in the instructions.

```
gdb lab09-2
~Register group: general~
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffcb0 0xffffcb0
ebp      0x0      0x0

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

native process 7650 (asm) In: _start L?? PC: 0x8049000
esi      0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
Quit
(gdb) x/1sb &msg1
0x804a000: "Hello, "
(gdb) x/1sb 0x804a008
0x804a008: "world!\n"
(gdb)
```

Рис. 4.12: Просмотр содержимого переменных двумя способами

Меняю содержимое переменных по имени и по адресу (рис. -fig. 4.13).



```

gdb lab09-2
~Register group: general~
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffcb0 0xffffcb0
ebp      0x0      0x0

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

native process 7650 (asm) In: _start      L??  PC: 0x8049000
'msg1' has unknown type; cast it to its declared type
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000: "hello, "
(gdb) set {char}&msg2='x'
(gdb) x/1sb &msg2
0x804a008: "xorld!\n"
(gdb)

```

Рис. 4.13: Изменение содержимого переменных двумя способами

Вывожу в различных форматах значение регистра edx (рис. -fig. 4.14).

```

gdb lab09-2
~Register group: general~
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffcb0 0xffffcb0
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0
eip      0x8049000 0x8049000 <_start>
eflags   0x202    [ IF ]


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 7650 (asm) In: _start      L??  PC: 0x8049000
(gdb) set {char}&msg2='x'
(gdb) x/1sb &msg2
0x804a008: "xorld!\n"
(gdb) p/t $ecx
$1 = 0
(gdb) p/t $edx
$2 = 0
(gdb) p/x $edx
$3 = 0x0
(gdb) p/s $edx
$4 = 0
(gdb)

```

Рис. 4.14: Просмотр значения регистра разными представлениями

С помощью команды set меняю содержимое регистра ebx (рис. -fig. 4.15).

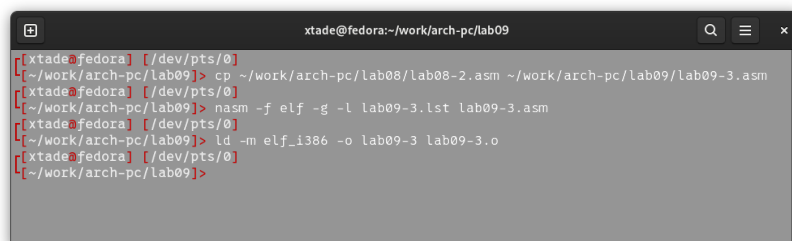


The screenshot shows the GDB interface for a process named 'gdb lab09-2'. The top panel displays the 'Register group: general' with values for registers: eax (0x0), ecx (0x0), edx (0x0), ebx (0x2), esp (0xffffcb0), ebp (0x0), esi (0x0), edi (0x0), eip (0x8049000), and eflags (0x202). The middle panel shows assembly code with addresses and instructions: 0x8049005: mov ebx, 0x1; 0x804900a: mov ecx, 0x804a000; 0x804900f: mov edx, 0x8; 0x8049014: int 0x80; 0x8049016: mov eax, 0x4; 0x804901b: mov ebx, 0x1; 0x8049020: mov ecx, 0x804a008; 0x8049025: mov edx, 0x7; 0x804902a: int 0x80. The bottom panel shows the 'native process 7650 (asm) In: \_start' with PC: 0x8049000. The command window shows the following commands and output: (gdb) p/s \$edx; 33 = 0x0; (gdb) set \$ebx='2'; (gdb) p/s; 34 = 0; (gdb) p/s \$ebx; 35 = 50; (gdb) set \$ebx=2; (gdb) p/s \$ebx; 36 = 2; (gdb)

Рис. 4.15: Примеры использования команды set

#### 4.1.4 Обработка аргументов командной строки в GDB

Копирую программу из предыдущей лабораторной работы в текущий каталог и создаю исполняемый файл с файлом листинга и отладки (рис. -fig. 4.16).



The screenshot shows a terminal window with the following commands and output: [xtade@fedora:~/work/arch-pc/lab09] cp ~/work/arch-pc/lab08/lab08-2.asm ~/work/arch-pc/lab09/lab09-3.asm; [xtade@fedora:~/work/arch-pc/lab09] nasm -f elf -g -l lab09-3.lst lab09-3.asm; [xtade@fedora:~/work/arch-pc/lab09] ld -m elf\_i386 -o lab09-3 lab09-3.o; [xtade@fedora:~/work/arch-pc/lab09]

Рис. 4.16: Подготовка новой программы

Запускаю программу с режиме отладки с указанием аргументов, указываю

брейкпоинт и запускаю отладку. Проверяю работу стека, изменяя аргумент команды просмотра регистра `esp` на `+4`, число обусловлено разрядностью системы, а указатель `void` занимает как раз 4 байта, ошибка при аргументе `+24` означает, что аргументы на вход программы закончились. (рис. -fig. 4.17).



```
gdb lab09-3
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab09-3...
(gdb) b _start
Breakpoint 1 at 0x80490e8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/xtade/work/arch-pc/lab09/lab09-3

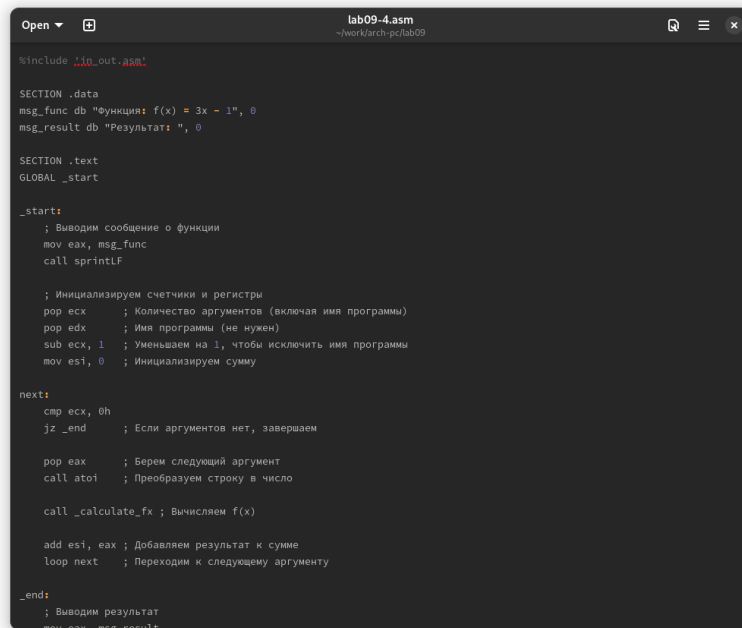
This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.

Breakpoint 1, _start () at lab09-3.asm:5
5      pop ecx ; Извлекаем из стека в 'ecx' количество
(gdb)
```

Рис. 4.17: Проверка работы стека

## 4.2 Задание для самостоятельной работы

1. Меняю программу самостоятельной части предыдущей лабораторной работы с использованием подпрограммы (рис. -fig. 4.18).



```
Open  lab09-4.asm
~\work\arch-ps\lab09

%include 'in_out.asm'

SECTION .data
msg_func db "Функция: f(x) = 3x - 1", 0
msg_result db "Результат: ", 0

SECTION .text
GLOBAL _start

_start:
; Выводим сообщение о функции
mov eax, msg_func
call sprintf

; Инициализируем счетчики и регистры
xor ecx, ecx ; Количество аргументов (включая имя программы)
xor edx, edx ; Имя программы (не нужен)
sub ecx, 1 ; Уменьшаем на 1, чтобы исключить имя программы
mov esi, 0 ; Инициализируем сумму

next:
cmp ecx, 0h
jz _end ; Если аргументов нет, завершаем

mov eax, [esp+ecx*4] ; Берем следующий аргумент
call atoi ; Преобразуем строку в число

call _calculate_fx ; Вычисляем f(x)

add esi, eax ; Добавляем результат к сумме
loop next ; Переходим к следующему аргументу

_end:
; Выводим результат
mov eax, msg_result
```

Рис. 4.18: Измененная программа предыдущей лабораторной работы

Код программы:

```
%include 'in_out.asm'

SECTION .data
msg_func db "Функция: f(x) = 3x - 1", 0
msg_result db "Результат: ", 0

SECTION .text
GLOBAL _start

_start:
    ; Выводим сообщение о функции
    mov eax, msg_func
    call sprintf
```

```

; Инициализируем счетчики и регистры
pop ecx      ; Количество аргументов (включая имя программы)
pop edx      ; Имя программы (не нужен)
sub ecx, 1    ; Уменьшаем на 1, чтобы исключить имя программы
mov esi, 0    ; Инициализируем сумму

next:
    cmp ecx, 0h
    jz _end    ; Если аргументов нет, завершаем

    pop eax    ; Берем следующий аргумент
    call atoi  ; Преобразуем строку в число

    call _calculate_fx ; Вычисляем  $f(x)$ 

    add esi, eax ; Добавляем результат к сумме
    loop next   ; Переходим к следующему аргументу

_end:
    ; Выводим результат
    mov eax, msg_result
    call sprint
    mov eax, esi
    call iprintLF

    ; Завершаем программу
    call quit

```

```

_calculate_fx:
    ; Вычисляем  $f(x) = 3x - 1$ 
    mov ebx, 3
    mul ebx
    sub eax, 1
    ret

```

Find More

2. Запускаю программу в режиме отладчика и пошагово через si просматриваю изменение значений регистров через i r. При выполнении инструкции mul ecx можно заметить, что результат умножения записывается в регистр eax, но также меняет и edx. Значение регистра ebx не обновляется напрямую, поэтому результат программа неверно подсчитывает функцию (рис. -fig. 4.19).

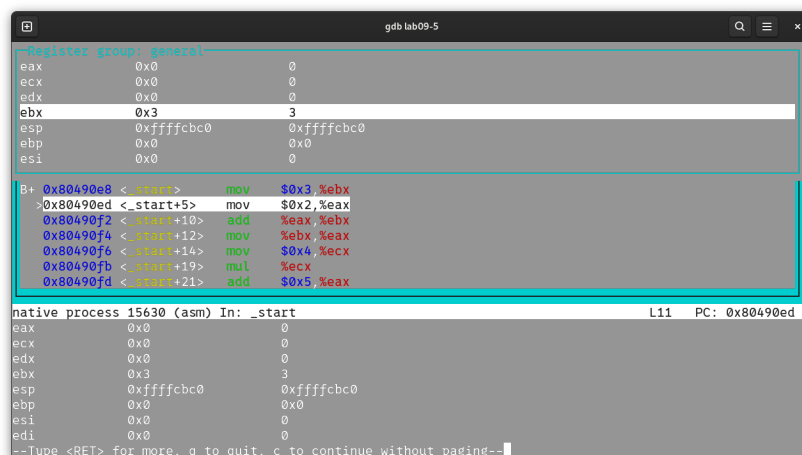


Рис. 4.19: Поиск ошибки в программе через пошаговую отладку

Исправляю найденную ошибку, теперь программа верно считает значение функции (рис. -fig. 4.20).



Рис. 4.20: Проверка корректировок в программе

Код измененной программы:

```
%include 'in_out.asm'
```

```
SECTION .data
```

```
div: DB 'Результат: ', 0
```

```
SECTION .text
```

```
GLOBAL _start
```

```
_start:
```

```
mov ebx, 3
```

```
mov eax, 2
```

```
add ebx, eax
```

```
mov 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
```



## **5 Выводы**

В результате выполнения данной лабораторной работы я приобрел навыки написания программ с использованием подпрограмм, а так же познакомился с методами отладки при помощи GDB и его основными возможностями.

## **6 Список литературы**

1. Курс на ТУИС
2. Лабораторная работа №9
3. Программирование на языке ассемблера NASM Столяров А. В.