

Отчёт по лабораторной работе 9

Архитектура компьютеров и операционные системы

Горелашвили Лия Михайловна НКАбд-03-23

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	9
4.1	Реализация подпрограмм в NASM	9
4.2	Отладка программ с помощью GDB	12
4.3	Задание для самостоятельной работы	23
5	Выводы	29

Список иллюстраций

4.1	Редактирование файла lab9-1.asm	10
4.2	Тестирование программы lab9-1.asm	10
4.3	Редактирование файла lab9-1.asm	11
4.4	Тестирование программы lab9-1.asm	12
4.5	Редактирование файла lab9-2.asm	13
4.6	Тестирование программы lab9-2.asm в отладчике	14
4.7	Дизассемблированный код	14
4.8	Дизассемблированный код в режиме интел	15
4.9	Точка остановки	16
4.10	Изменение регистров	17
4.11	Изменение регистров	18
4.12	Изменение значения переменной	19
4.13	Вывод значения регистра	20
4.14	Вывод значения регистра	21
4.15	Вывод значения регистра	22
4.16	Редактирование файла prog-1.asm	23
4.17	Тестирование программы prog-1.asm	24
4.18	Код с ошибкой	25
4.19	Отладка	26
4.20	Код исправлен	27
4.21	Проверка работы	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 Задание

1. Изучение подпрограмм в ассемблере
2. Освоение возможностей отладчика GDB
3. Рассмотрение примеров работы с отладчиком
4. Выполнение заданий для самостоятельной работы

3 Теоретическое введение

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.

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. Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

Для вызова подпрограммы из основной программы используется инструкция `call`, которая заносит адрес следующей инструкции в стек и загружает в регистр `esp` адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы.

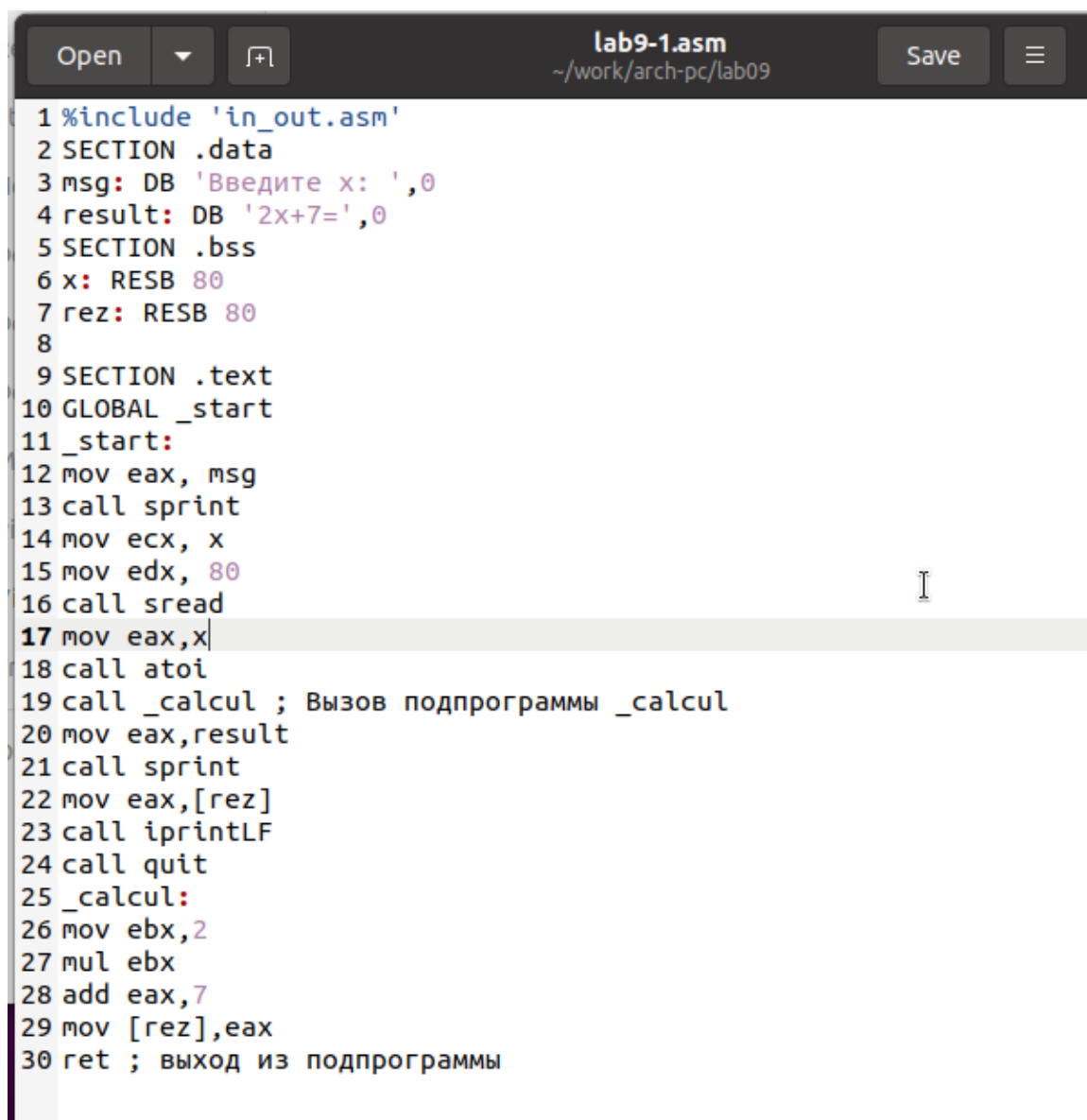
Подпрограмма завершается инструкцией `ret`, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией `call`, и заносит его в `esp`.

После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией call.

4 Выполнение лабораторной работы

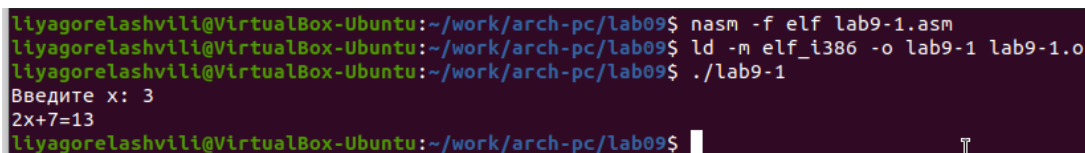
4.1 Реализация подпрограмм в NASM

Для начала я создала новую директорию и перешла в нее, чтобы выполнить лабораторную работу номер 9. Затем создала файл с именем `lab9-1.asm`, в котором реализовала программу для вычисления арифметического выражения $f(x) = 2x + 7$ с использованием подпрограммы `calcul`. Для этого я вводила значение переменной `x` с клавиатуры, а само выражение вычислялось внутри подпрограммы.



```
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
```

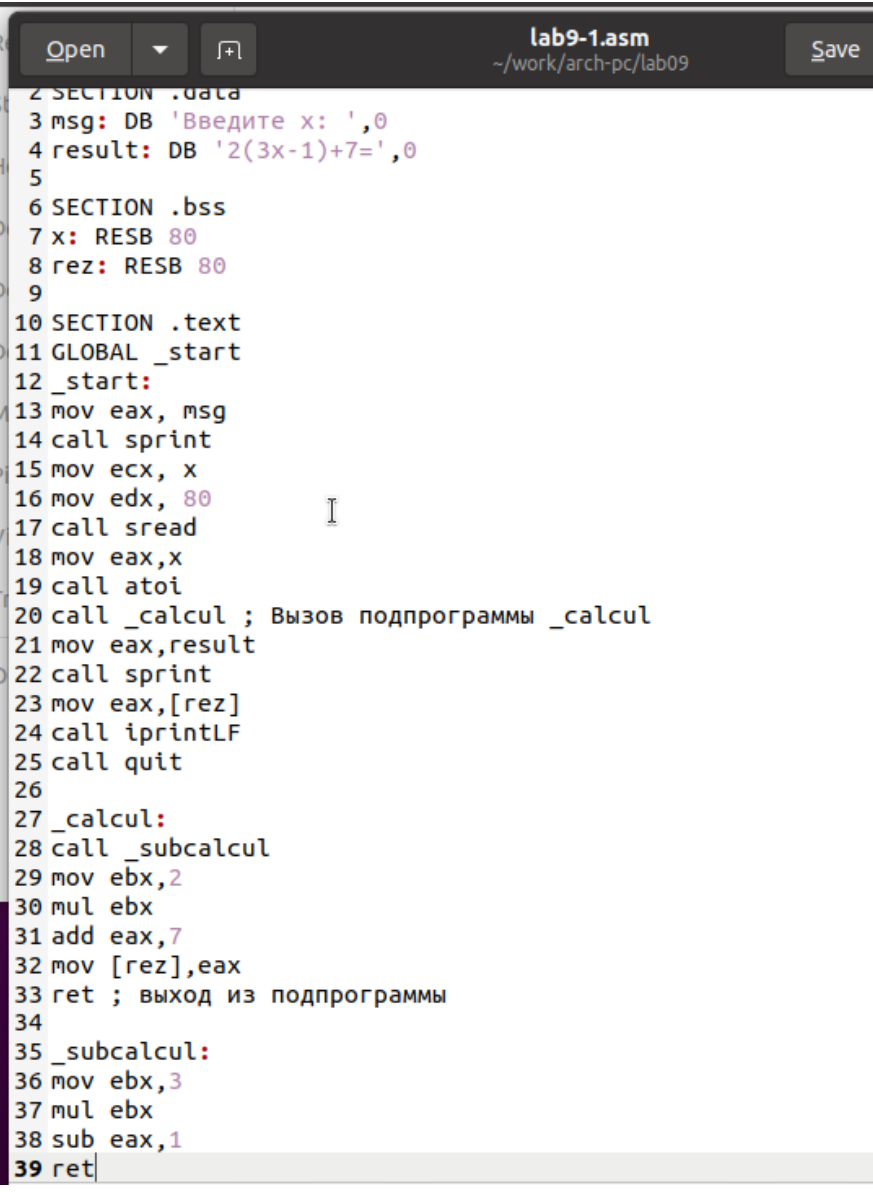
Рис. 4.1: Редактирование файла lab9-1.asm



```
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 3
2x+7=13
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$
```

Рис. 4.2: Тестирование программы lab9-1.asm

После этого я внесла изменения в текст программы, добавив подпрограмму `subcalcul` внутрь подпрограммы `calcul`. Это позволяет вычислить составное выражение $f(g(x))$, где значение x также вводится с клавиатуры. Функции определены следующим образом: $f(x) = 2x + 7$, $g(x) = 3x - 1$.



```

1  Open  ▾  [+]
```

```

2 SECTION .data
3 msg: DB 'Введите x: ',0
4 result: DB '2(3x-1)+7=',0
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. 4.3: Редактирование файла lab9-1.asm

```
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 3
2x+7=13
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 3
2(3x-1)+7=23
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$
```

Рис. 4.4: Тестирование программы lab9-1.asm

4.2 Отладка программ с помощью GDB

Также я создала файл с именем lab9-2.asm, в котором содержится программа для вывода сообщения “Hello world!”. Я скомпилировала этот файл и получила исполняемый файл. Чтобы добавить отладочную информацию для работы с отладчиком GDB, я использовала ключ “-g”.

```
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. 4.5: Редактирование файла lab9-2.asm

Затем я загрузила полученный исполняемый файл в отладчик GDB и проверила его работу, запустив программу с помощью команды “run” или “r”. Чтобы получить более детальный анализ программы, я установила точку остановки на метке “start”, с которой начинается выполнение любой ассемблерной программы, и запустила ее. После этого я просмотрела дизассемблированный код программы.

```

liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ nasm -f elf -g -l lab9-2.lst lab9-2.asm
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ gdb lab9-2
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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) run
Starting program: /home/liyagorelashvili/work/arch-pc/lab09/lab9-2
Hello, world!
[Inferior 1 (process 3364) exited normally]
(gdb)

```

Рис. 4.6: Тестирование программы lab9-2.asm в отладчике

```

[Inferior 1 (process 3364) exited normally]
(gdb) break _start
Breakpoint 1 at 0x08049000
(gdb) run
Starting program: /home/liyagorelashvili/work/arch-pc/lab09/lab9-2

Breakpoint 1, 0x08049000 in _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. 4.7: Дизассемблированный код

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09
(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. 4.8: Дизассемблированный код в режиме интел

Для проверки точки останова по имени метки “_start” я воспользовалась командой “info breakpoints” или “i b”. Кроме того, я установила еще одну точку останова по адресу инструкции, определив адрес предпоследней инструкции “mov ebx, 0x0”. Это помогло мне контролировать выполнение программы и анализировать ее состояние в отладчике GDB.

```

liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09

Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd1a0 0xffffd1a0
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 3368 In: _start L?? PC
(gdb)
(gdb)
(gdb) b *0x8049031Breakpoint 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: Точка остановки

В GDB я имею возможность просматривать содержимое ячеек памяти и регистров, а также изменять значения регистров и переменных. Для отслеживания изменений значений регистров, использовала команду 'stepi' (сокращенно 'si'), которая позволяет выполнить одну инструкцию за раз. Это позволило мне следить за состоянием программы и анализировать изменения регистров.


```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09

Register group: general
eax      0x4      4
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd1a0 0xffffd1a0
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 3368 In: _start L?? PC: 0x8049005
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
0x8049005 in _start ()
(gdb) 
```

Рис. 4.10: Изменение регистров

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd1a0 0xffffd1a0
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 3368 In: _start L?? PC: 0x8049016
(gdb) si
0x08049005 in _start ()
(gdb) si
0x0804900a in _start ()
(gdb) si
0x0804900f in _start ()
(gdb) si
0x08049014 in _start ()
(gdb) si
0x08049016 in _start ()
(gdb) 
```

Рис. 4.11: Изменение регистров

Для просмотра значения переменной `msg1` по имени и получения нужных данных, использовала соответствующую команду, предоставленную отладчиком GDB.

Еще одной полезной командой была команда `set`, которую я использовала для изменения значения регистра или ячейки памяти. Я указывала имя регистра или адрес в качестве аргумента команды `set`, и успешно изменяла значения переменных и регистров в процессе отладки программы.

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd1a0 0xffffd1a0
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 3368 In: _start L?? PC: 0x8049016
(gdb)
(gdb) x/1sb &msg10x804a000 <msg1>:      "Hello, "
(gdb)
(gdb) x/1sb 0x804a0080x804a008 <msg2>:   "world!\n"
(gdb)
(gdb)
(gdb) x/1sb &msg10x804a000 <msg1>:      "hello, "
(gdb)
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "Lorld!\n"
(gdb) █
```

Рис. 4.12: Изменение значения переменной

В частности, я успешно изменила первый символ переменной msg1, что позволило мне проверить поведение программы при изменении данных.

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09

Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x1      1
esp      0xffffd1a0 0xffffd1a0
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 3368 In: start L?? PC: 0x8049016
(gdb)
(gdb) p/s $ecx$3 = 134520832
(gdb)
(gdb) p/x $ecx$4 = 0x804a000
(gdb)
(gdb) p/s $edx$5 = 8
(gdb)
(gdb) p/t $edx$6 = 1000
(gdb) p/x $edx
$7 = 0x8
(gdb)
```

Рис. 4.13: Вывод значения регистра

Также, с помощью команды `set`, я изменяла значение регистра `ebx` на нужное значение, чтобы проверить влияние такой модификации на выполнение программы.

The screenshot shows a GDB terminal window with the title bar "liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09". The window is divided into three main sections. The top section, titled "Register group: general", lists the values of general-purpose registers: `eax` (0x8), `ecx` (0x804a000), `edx` (0x8), `ebx` (0x2), `esp` (0xffffd1a0), `ebp` (0x0), `esi` (0x0), `edi` (0x0), and `eip` (0x8049016). The middle section displays assembly code with addresses and instructions, such as `B+ 0x8049000 <_start> mov eax,0x4` and `>0x8049016 <_start+22> mov eax,0x4`. The bottom section shows GDB commands and their output: `(gdb) p/t edx6 = 1000`, `(gdb) p/x $edx` resulting in `$7 = 0x8`, `(gdb) p/s ebx8 = 50`, and `(gdb) p/s $ebx` resulting in `$9 = 2`. The status bar at the bottom indicates "native process 3368 In: _start" and "PC: 0x8049016".

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09
Register group: general
eax      0x8      8
ecx      0x804a000 134520832
edx      0x8      8
ebx      0x2      2
esp      0xffffd1a0 0xffffd1a0
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 3368 In: _start      L??  PC: 0x8049016
(gdb)
(gdb) p/t $edx$6 = 1000
(gdb) p/x $edx
$7 = 0x8
(gdb)
(gdb)
(gdb) p/s $ebx$8 = 50
(gdb)
(gdb) p/s $ebx
$9 = 2
(gdb)
```

Рис. 4.14: Вывод значения регистра

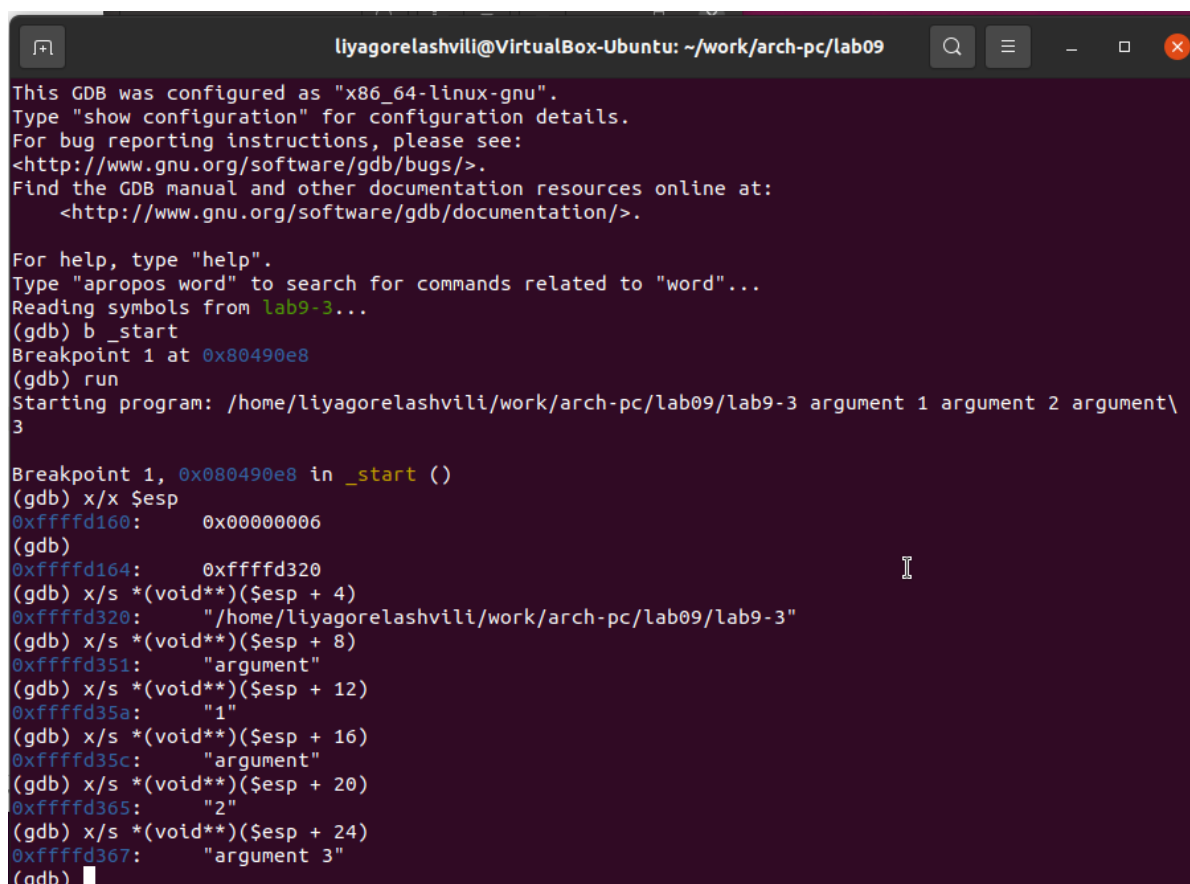
Для выполнения лабораторной работы, я решила использовать файл `lab8-2.asm`, который был создан в процессе выполнения предыдущей лабораторной работы №8. Этот файл содержит программу, которая выводит аргументы командной строки. Для начала, я скопировала этот файл и создала исполняемый файл из скопированного исходного файла.

Для загрузки программы с аргументами в отладчик GDB, я использовала ключ `-args` и загрузила исполняемый файл в отладчик с указанными аргументами. Затем, установила точку останова перед первой инструкцией программы и запустила ее.

В процессе отладки, я обратила внимание на адрес вершины стека, который хранится в регистре `esp`. По этому адресу, обнаружила число, указывающее ко-

личество аргументов командной строки. В данном случае, количество аргументов равно 5, включая имя программы lab9-3 и аргументы: аргумент1, аргумент2 и 'аргумент 3'.

Далее, я просмотрела остальные позиции стека. По адресу [esp+4], нашла адрес в памяти, где располагается имя программы. По адресу [esp+8] хранится адрес первого аргумента, по адресу [esp+12] - второго, и так далее. Шаг изменения адреса равен 4 байта, так как каждый следующий адрес на стеке находится на расстоянии 4 байт от предыдущего ([esp+4], [esp+8], [esp+12]).



```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

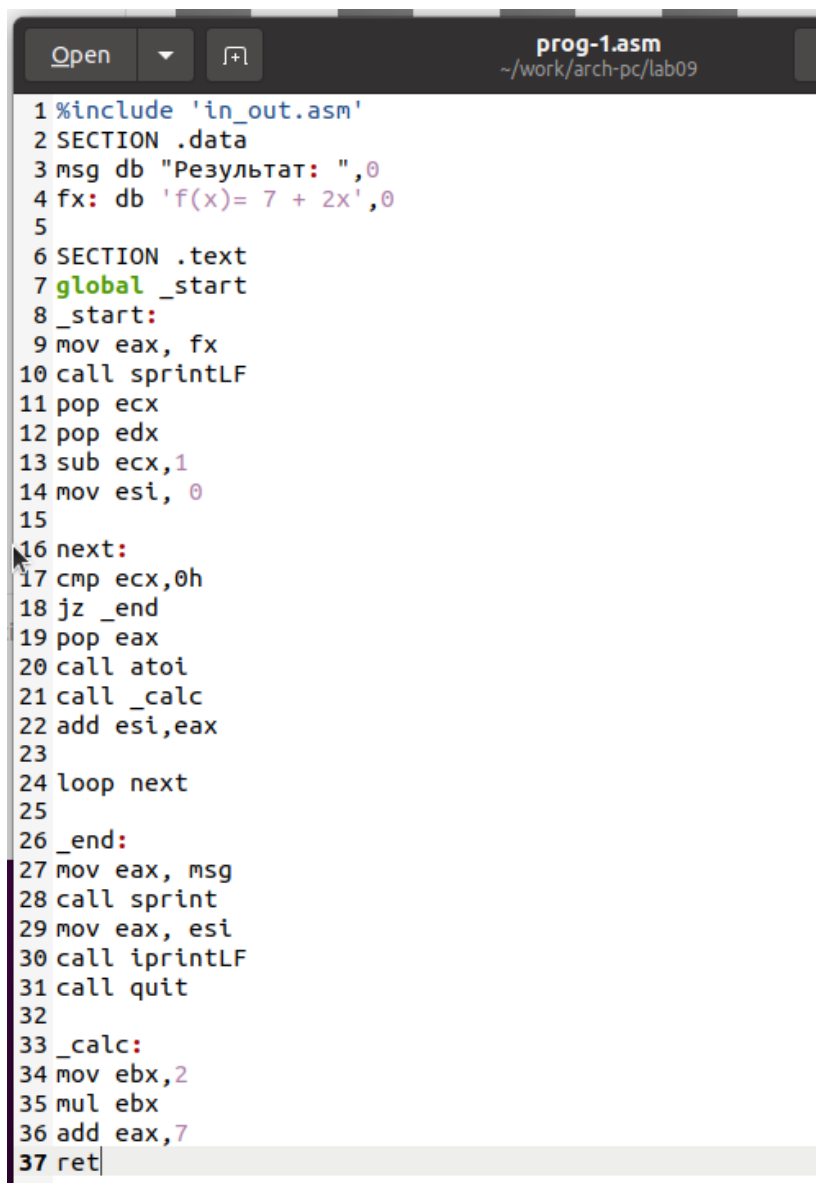
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab9-3...
(gdb) b _start
Breakpoint 1 at 0x080490e8
(gdb) run
Starting program: /home/liyagorelashvili/work/arch-pc/lab09/lab9-3 argument 1 argument 2 argument\
3

Breakpoint 1, 0x080490e8 in _start ()
(gdb) x/x $esp
0xffffd160: 0x00000006
(gdb)
0xffffd164: 0xffffd320
(gdb) x/s *(void**)($esp + 4)
0xffffd320: "/home/liyagorelashvili/work/arch-pc/lab09/lab9-3"
(gdb) x/s *(void**)($esp + 8)
0xffffd351: "argument"
(gdb) x/s *(void**)($esp + 12)
0xffffd35a: "1"
(gdb) x/s *(void**)($esp + 16)
0xffffd35c: "argument"
(gdb) x/s *(void**)($esp + 20)
0xffffd365: "2"
(gdb) x/s *(void**)($esp + 24)
0xffffd367: "argument 3"
(gdb)
```

Рис. 4.15: Вывод значения регистра

4.3 Задание для самостоятельной работы

Я переписала программу из лабораторной работы №8, задание №1, чтобы реализовать вычисление значения функции $f(x)$ как подпрограмму.



```
1 %include 'in_out.asm'
2 SECTION .data
3 msg db "Результат: ",0
4 fx: db 'f(x)= 7 + 2x',0
5
6 SECTION .text
7 global _start
8 _start:
9 mov eax, fx
10 call sprintLF
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 sprint
29 mov eax, esi
30 call iprintLF
31 call quit
32
33 _calc:
34 mov ebx,2
35 mul ebx
36 add eax,7
37 ret
```

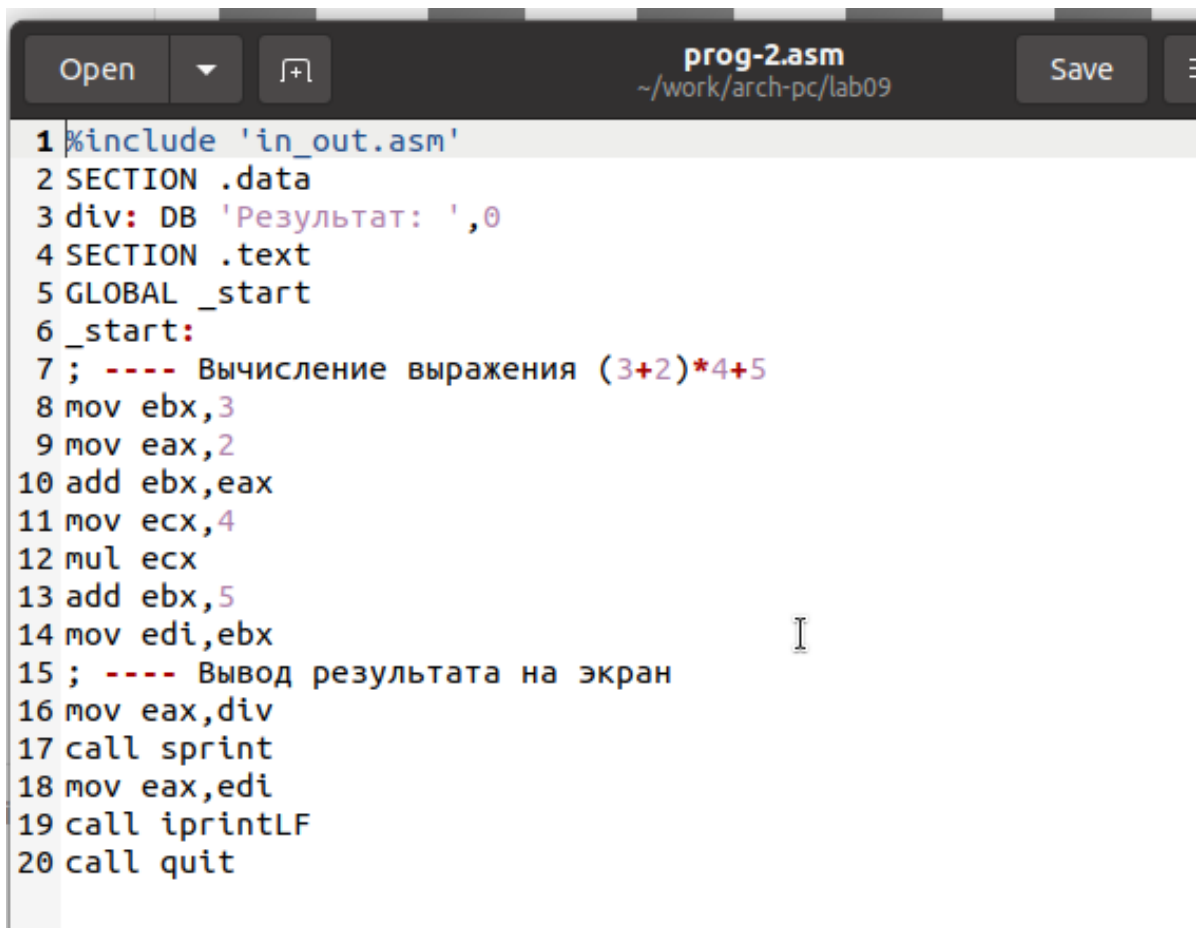
Рис. 4.16: Редактирование файла prog-1.asm

```
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ nasm -f elf prog-1.asm
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ld -m elf_i386 -o prog-1 prog-1.o
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ./prog-1 1
f(x)= 7 + 2x
Результат: 9
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$ ./prog-1 1 3 6 4
f(x)= 7 + 2x
Результат: 56
liyagorelashvili@VirtualBox-Ubuntu:~/work/arch-pc/lab09$
```

Рис. 4.17: Тестирование программы prog-1.asm

Приведенный ниже код представляет программу для вычисления выражения $(3 + 2) * 4 + 5$. Однако, при запуске, программа дает неверный результат.

Я провела анализ изменений значений регистров с помощью отладчика GDB и обнаружила ошибку: перепутан порядок аргументов у инструкции add. Также заметила, что по окончании работы программы в регистр edi передается значение ebx вместо eax.



```
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. 4.18: Код с ошибкой

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09

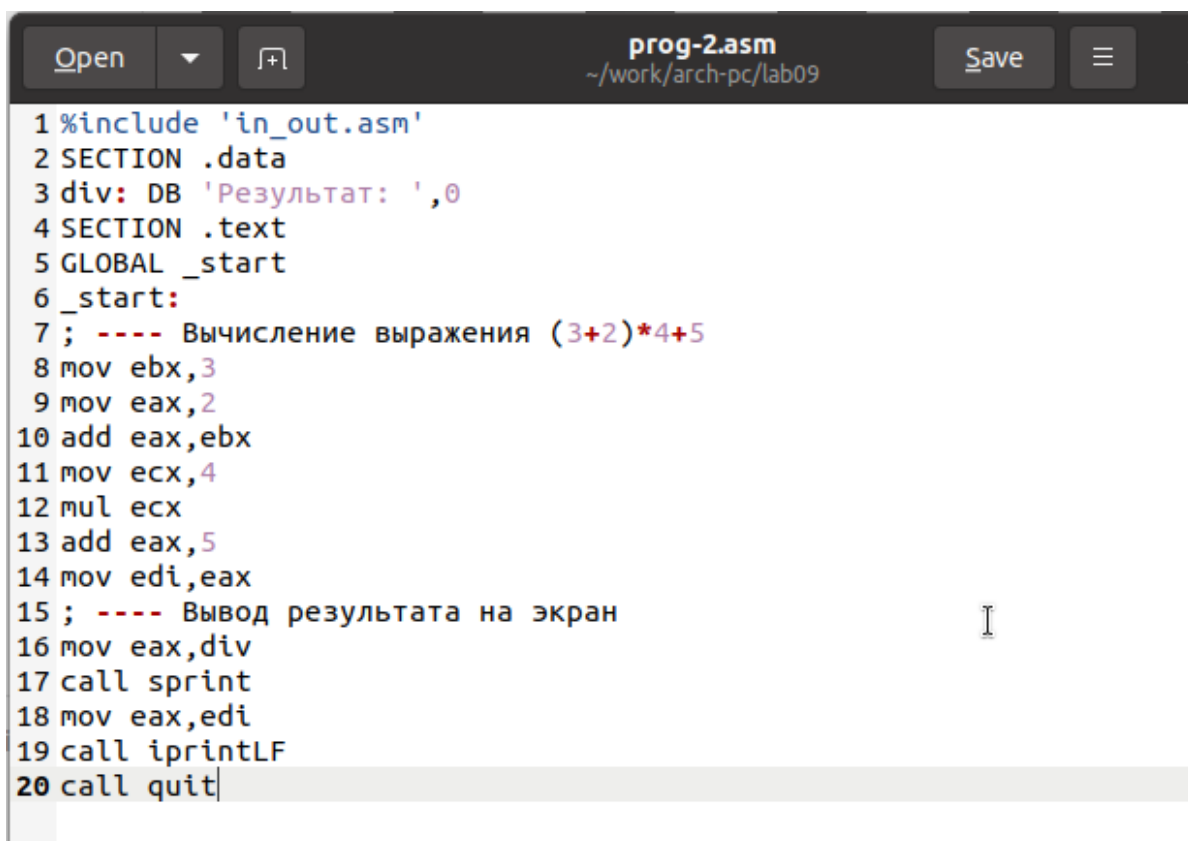
eax      0x8      8
ecx      0x4      4
edx      0x0      0
ebx      0xa      10
esp      0xffffd1a0 0xffffd1a0
ebp      0x0      0x0
esi      0x0      0
edi      0xa      10
eip      0x8049100 0x8049100 <_start+24>

B+ 0x80490e8 <_start>      mov     ebx,0x3
B+ 0x80490e8 <_start>5>    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,0x5
0x80490fb <_start>19>     add     ebx,0x5
>0x80490fe <_start>22>     mov     edi,ebx04a000
0x8049100 <_start>24>     mov     eax,0x804a000rint>
0x8049105 <_start>29>     call   0x804900f <sprint>
0x804910a <_start>34>     mov     eax,edi

native process 3998 In: _start      L??  PC: 0x8049100
(gdb) sNo process In:              L??  PC: ??
(gdb) si
0x080490fb in _start ()
(gdb) si
0x080490fe in _start ()
(gdb) si
0x08049100 in _start ()
(gdb) c
Continuing.
Результат: 10
[Inferior 1 (process 3998) exited normally]
(gdb) 
```

Рис. 4.19: Отладка

Я внесла необходимые исправления в код программы, учитывая перепутанный порядок аргументов у инструкции `add` и правильную передачу значения в регистр `edi` по окончании работы программы. Это позволило исправить ошибку и получить правильный результат вычисления выражения.



```
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. 4.20: Код исправлен

```
liyagorelashvili@VirtualBox-Ubuntu: ~/work/arch-pc/lab09
eax      0x19      25
ecx      0x4       4
edx      0x0       0
ebx      0x3       3
esp      0xffffd1a0 0xffffd1a0
ebp      0x0       0
esi      0x0       0
edi      0x19      25
eip      0x8049100 0x8049100 <_start+24>

B+ 0x80490e8 <_start>      mov     ebx,0x3
B+ 0x80490e8 <_start+5>    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,0x5
0x80490fb <_start+19>     add     eax,0x5
>0x80490fe <_start+22>    mov     edi,eax04a000
0x8049100 <_start+24>    mov     eax,0x804a000rint>
0x8049105 <_start+29>    call   0x804900f <sprint>
0x804910a <_start+34>    mov     eax,edi

native process 4100 In: _start      L??  PC: 0x8049100
(gdb) sNo process In:              L??  PC: ??
(gdb) si
0x080490fb in _start ()
(gdb) si
0x080490fe in _start ()
(gdb) si
0x08049100 in _start ()
(gdb) c
Continuing.
Результат: 25
[Inferior 1 (process 4100) exited normally]
(gdb) █
```

Рис. 4.21: Проверка работы

5 Выводы

Освоили работу с подпрограммами и отладчиком.