

Отчёт по лабораторной работе №9

Дисциплина: Архитектура Компьютера

Егор Витальевич Кузьмин

Содержание

1	Цель работы	4
2	Задание	5
3	Теоретическое введение	6
4	Выполнение лабораторной работы	10
5	Выводы	27
	Список литературы	28

Список иллюстраций

4.1	Работа с директориями и копирование, создание, редактирование файла	11
4.2	Создание и запуск исполняемого файла	11
4.3	Редактирование файла	12
4.4	Создание и запуск исполняемого файла	12
4.5	Редактирование файла	13
4.6	Создание исполняемого файла, отладчик gdb	13
4.7	Отладчик gdb	13
4.8	Установка метки	13
4.9	Работа метки	14
4.10	Дисассимилированный код программы с синтаксисом intel	14
4.11	Режим псевдографики	15
4.12	Режим псевдографики	15
4.13	Проверка точек останова	16
4.14	Редактирование файла	16
4.15	Создание и запуск исполняемого файла	16
4.16	Изменение значений регистров и переменных	16
4.17	Изменение значений регистров и переменных	17
4.18	Содержимое регистров	17
4.19	Содержимое переменной, изменение в ней символов	18
4.20	Содержимое переменной, изменение в ней символов	18
4.21	Значения регистра edx	19
4.22	Изменение значений регистра	19
4.23	Копирование файла, создание исполняемого файла	20
4.24	Загрузка файла в отладчик	20
4.25	Установка точки останова, запуск программы	20
4.26	Просмотр позиций стека	21
4.27	Копирование, редактирование файла	21
4.28	Создание и запуск исполняемого файла	22
4.29	Редактирование файла	23
4.30	Создание и запуск исполняемого файла	23

1 Цель работы

Целью данной работы является приобретение практического опыта в написании программ с использованием подпрограмм, а также знакомство с методами отладки при помощи gdb и его основными возможностями.

2 Задание

1. Реализация подпрограмм в NASM.
2. Отладка программ при помощи gdb.
3. Выполнение заданий для самостоятельной 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, отработывает, 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.

Наиболее часто применяют следующие методы отладки:

- создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые диагностические сообщения);
- использование специальных программ-отладчиков.

Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять данные. Это помогает быстрее найти место ошибки в программе и ускорить её исправление. Наиболее популярные способы работы с отладчиком — это использование точек останова и выполнение программы по шагам. Пошаговое выполнение — это выполнение программы с остановкой после каждой строки, чтобы программист мог проверить значения переменных и выполнить другие действия. Точки останова — это специально отмеченные места в программе, в которых программа-отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды точек останова:

- Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмеченной программистом);
- Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его).

Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы. GDB (GNU Debugger — отладчик проекта GNU) [1] 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. GDB может выполнять следующие действия:

- начать выполнение программы, задав всё, что может повлиять на её поведение;
- остановить программу при указанных условиях;
- исследовать, что случилось, когда программа остановилась;
- изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

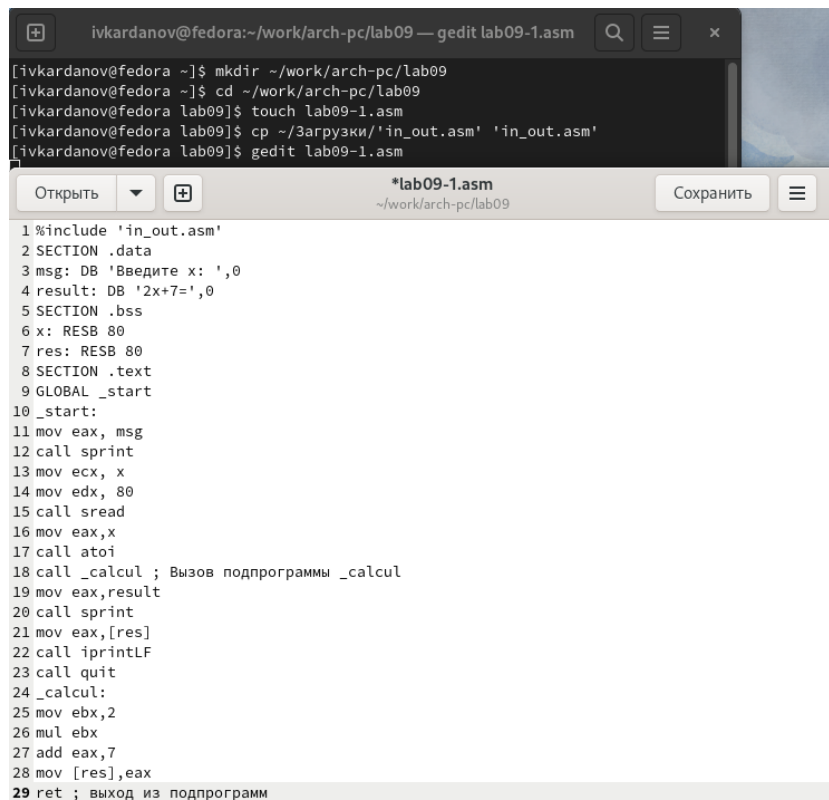
После запуска `gdb` выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (`gdb`) для ввода команд. Далее приведён список некоторых команд GDB. Команда `run` (сокращённо `r`) — запускает отлаживаемую программу в оболочке GDB. Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др. Команда `kill` (сокращённо `k`) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки. Если в ответ введено `y` (то есть «да»), отладка программы прекращается. Командой `run` её можно начать заново, при этом все точки останова (`breakpoints`), точки просмотра (`watchpoints`) и точки отлова (`catchpoints`) сохраняются. Для выхода из отладчика используется команда `quit` (или сокращённо `q`). Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом `-g`. Установить точку останова можно командой `break` (кратко `b`). Типич-

ный аргумент этой команды — место установки. Его можно задать как имя метки или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка». Для продолжения остановленной программы используется команда `continue (c)` (gdb). Выполнение программы будет происходить до следующей точки останова. В качестве аргумента может использоваться целое число \star , которое указывает отладчику проигнорировать $\star - 1$ точку останова (выполнение остановится на \star -й точке). Команда `stepi` (кратко `sI`) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию. Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных. Посмотреть содержимое регистров можно с помощью команды `info registers` (или `i r`). 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.

С помощью утилиты `mkdir` создаю директорию `lab09` для выполнения соответствующей лабораторной работы. Перехожу в созданный каталог с помощью утилиты `cd`. С помощью `touch` создаю файл `lab09-1.asm`. Копирую в текущий каталог файл `in_out.asm` с помощью утилиты `cp`, ибо он будет использоваться в дальнейшем. Открываю созданный файл `lab09-1.asm`, вставляю в него следующую программу: (рис. 4.1).



The screenshot shows a terminal window with the following commands and output:

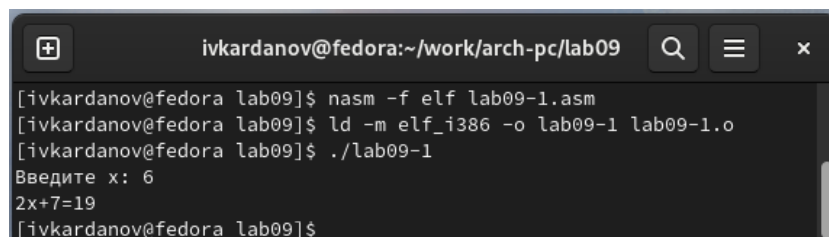
```
ivkardanov@fedora:~/work/arch-pc/lab09 — gedit lab09-1.asm
[ivkardanov@fedora ~]$ mkdir ~/work/arch-pc/lab09
[ivkardanov@fedora ~]$ cd ~/work/arch-pc/lab09
[ivkardanov@fedora lab09]$ touch lab09-1.asm
[ivkardanov@fedora lab09]$ cp ~/3арязыки/'in_out.asm' 'in_out.asm'
[ivkardanov@fedora lab09]$ gedit lab09-1.asm
```

The editor window shows the following assembly code:

```
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 res: RESB 80
8 SECTION .text
9 GLOBAL _start
10 _start:
11 mov eax, msg
12 call sprint
13 mov ecx, x
14 mov edx, 80
15 call sread
16 mov eax, x
17 call atoi
18 call _calcul ; Вызов подпрограммы _calcul
19 mov eax, result
20 call sprint
21 mov eax, [res]
22 call iprintLF
23 call quit
24 _calcul:
25 mov ebx, 2
26 mul ebx
27 add eax, 7
28 mov [res], eax
29 ret ; выход из подпрограмм
```

Рис. 4.1: Работа с директориями и копирование, создание, редактирование файла

Создаю исполняемый файл и запускаю его. (рис. 4.2).

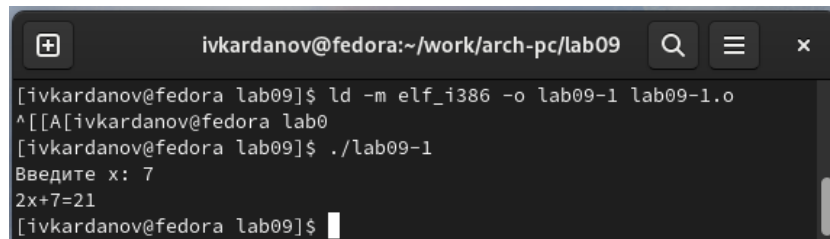


The screenshot shows a terminal window with the following commands and output:

```
ivkardanov@fedora:~/work/arch-pc/lab09
[ivkardanov@fedora lab09]$ nasm -f elf lab09-1.asm
[ivkardanov@fedora lab09]$ ld -m elf_i386 -o lab09-1 lab09-1.o
[ivkardanov@fedora lab09]$ ./lab09-1
Введите x: 6
2x+7=19
[ivkardanov@fedora lab09]$
```

Рис. 4.2: Создание и запуск исполняемого файла

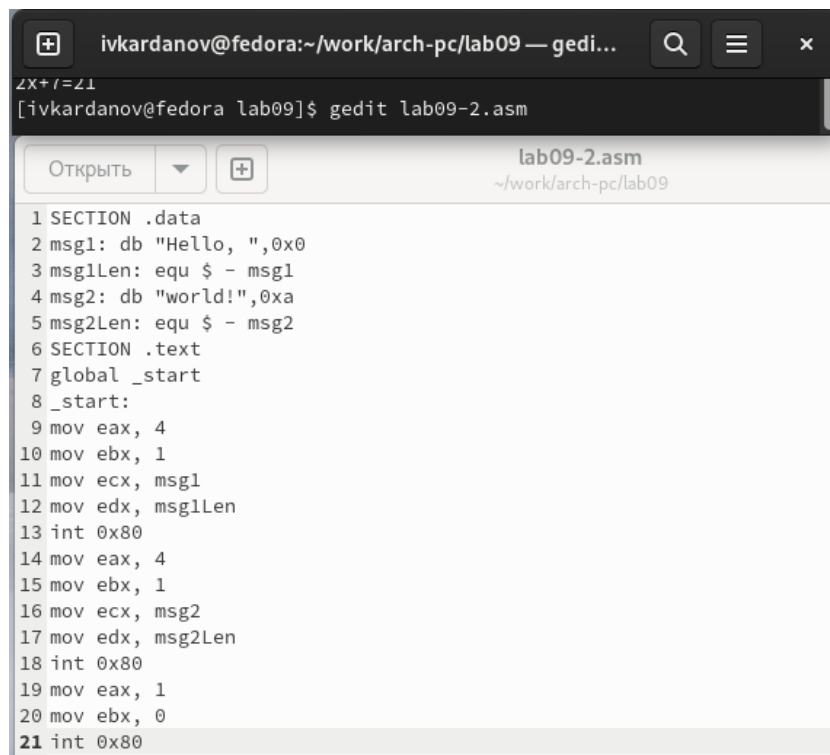
Добавляю подпрограмму `subcalcul_`, чтобы программа вычисляла значение $f(g(x))$. (рис. 4.3).



```
ivkardanov@fedora:~/work/arch-pc/lab09
[ivkardanov@fedora lab09]$ ld -m elf_i386 -o lab09-1 lab09-1.o
^[A[ivkardanov@fedora lab0
[ivkardanov@fedora lab09]$ ./lab09-1
Введите x: 7
2x+7=21
[ivkardanov@fedora lab09]$
```

Рис. 4.3: Редактирование файла

Создаю исполняемый файл и убеждаюсь в правильности его работы. (рис. 4.4).



```
ivkardanov@fedora:~/work/arch-pc/lab09 — gedi...
2x+7=21
[ivkardanov@fedora lab09]$ gedit lab09-2.asm

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

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

Рис. 4.4: Создание и запуск исполняемого файла

4.2) Отладка программ при помощи gdb.

Создаю файл lab09-2.asm и вношу в него следующий текст программы: (рис. 4.5).

```
ivkardanov@fedora:~/work/arch-pc/lab09 — gdb lab09-2
[ivkardanov@fedora lab09]$ nasm -f elf -g -l lab09-2.lst lab09-2.asm
[ivkardanov@fedora lab09]$ ld -m elf_i386 -o lab09-2 lab09-2.o
[ivkardanov@fedora lab09]$ gdb lab09-2
GNU gdb (GDB) Fedora Linux 13.2-6.fc38
Copyright (C) 2023 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"...
--Type <RET> for more, q to quit, c to continue without paging--c
Reading symbols from lab09-2...
(gdb) run
Starting program: /home/ivkardanov/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])
```

Рис. 4.5: Редактирование файла

Создаю исполняемый файл и загружаю его в отладчик gdb, запускаю программу с помощью команды run. (рис. 4.6).

```
Starting program: /home/ivkardanov/work/arch-pc/lab09/lab09-2
Hello, world!
```

Рис. 4.6: Создание исполняемого файла, отладчик gdb

Убеждаюсь в правильности работы программы. (рис. 4.7).

```
(gdb) break _start
Breakpoint 1 at 0x4010e0: file lab09-2.asm, line 9.
```

Рис. 4.7: Отладчик gdb

Устанавливаю метку _start. (рис. 4.8).

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

Breakpoint 1, _start () at lab09-2.asm:9
9      mov eax, 4
```

Рис. 4.8: Установка метки

Запускаю программу, видим работу метки. (рис. 4.9).

```

9      mov eax, 4
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x004010e0 <+0>:   mov     $0x4,%eax
0x004010e5 <+5>:     mov     $0x1,%ebx
0x004010ea <+10>:    mov     $0x402118,%ecx
0x004010ef <+15>:    mov     $0xb,%edx
0x004010f4 <+20>:    int     $0x80
0x004010f6 <+22>:    mov     $0x4,%eax
0x004010fb <+27>:    mov     $0x1,%ebx
0x00401100 <+32>:    mov     $0x402120,%ecx
0x00401105 <+37>:    mov     $0x7,%edx
0x0040110a <+42>:    int     $0x80
0x0040110c <+44>:    mov     $0x1,%eax
0x00401111 <+49>:    mov     $0xb,%ebx
0x00401116 <+54>:    int     $0x80

End of assembler dump.
(gdb) set disassembler flavor intel
Ambiguous set command "disassembler flavor intel": disassemble-next-line, disassembler-options.
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x004010e0 <+0>:   mov     $0x4,%eax
0x004010e5 <+5>:     mov     $0x1,%ebx
0x004010ea <+10>:    mov     $0x402118,%ecx
0x004010ef <+15>:    mov     $0xb,%edx
0x004010f4 <+20>:    int     $0x80
0x004010f6 <+22>:    mov     $0x4,%eax
0x004010fb <+27>:    mov     $0x1,%ebx
0x00401100 <+32>:    mov     $0x402120,%ecx
0x00401105 <+37>:    mov     $0x7,%edx
0x0040110a <+42>:    int     $0x80
0x0040110c <+44>:    mov     $0x1,%eax
0x00401111 <+49>:    mov     $0xb,%ebx
0x00401116 <+54>:    int     $0x80

End of assembler dump.
(gdb)

```

Рис. 4.9: Работа метки

Смотрю дисассимилированный код программы сначала обычный, потом с синтаксисом intel. (рис. 4.10).

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

```

B> 0x4010e0 <_start>      mov     $0x4,%eax
0x4010e5 <_start+5>      mov     $0x1,%ebx
0x4010ea <_start+10>     mov     $0x402118,%ecx
0x4010ef <_start+15>     mov     $0x8,%edx
0x4010f4 <_start+20>     int     $0x80
0x4010f6 <_start+22>     mov     $0x4,%eax
0x4010fb <_start+27>     mov     $0x1,%ebx
0x401109 <_start+32>     mov     $0x402120,%ecx
0x401105 <_start+37>     mov     $0x7,%edx
0x40110a <_start+42>     int     $0x80
0x40110c <_start+44>     mov     $0x1,%eax
0x401111 <_start+49>     mov     $0xc,%ebx
0x401116 <_start+54>     int     $0x80
0x401118                      dec     %eax
0x401119                      gs insb (%dx),%es:(%edi)
0x40111b                      insb   (%dx),%es:(%edi)
0x40111c                      outsl  %ds:(%esi),(%dx)
0x40111d                      sub    $0x20,%al
0x40111f                      add    %di,%bfi(%edi)
0x401122                      jb     0x401190
0x401124                      and    %ecx,%fs:(%edx)

```

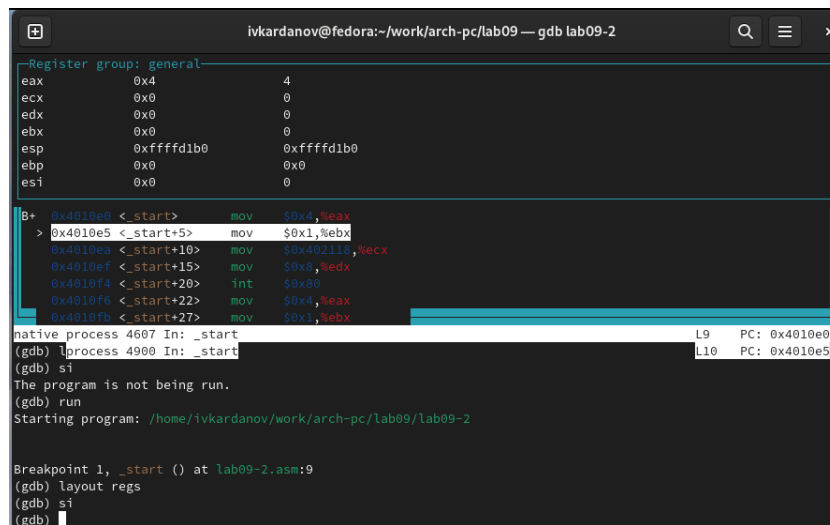
native process 4607 In: _start L9 PC: 0x4010e0

(gdb)

Рис. 4.10: Дисассимилированный код программы с синтаксисом intel

Различия отображения синтаксиса можно наблюдать в правой части окна.

Затем я включаю режим псевдографики (рис. 4.11).



```
Register group: general
eax      0x4      4
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd1b0 0xffffd1b0
ebp      0x0      0
esi      0x0      0

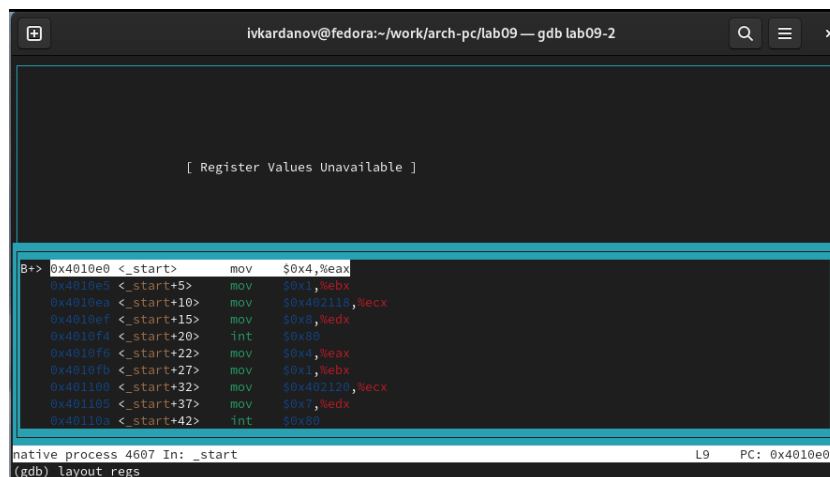
B+ 0x4010e0 <_start> mov $0x4,%eax
> 0x4010e5 <_start+5> mov $0x1,%ebx
0x4010ea <_start+10> mov $0x402118,%ecx
0x4010ef <_start+15> mov $0x8,%edx
0x4010f4 <_start+20> int $0x80
0x4010f6 <_start+22> mov $0x4,%eax
0x4010fb <_start+27> mov $0x1,%ebx

native process 4607 In: _start L9 PC: 0x4010e0
(gdb) si L10 PC: 0x4010e5
(gdb) si
The program is not being run.
(gdb) run
Starting program: /home/ivkardanov/work/arch-pc/lab09/lab09-2

Breakpoint 1, _start () at lab09-2.asm:9
(gdb) layout regs
(gdb) si
(gdb)
```

Рис. 4.11: Режим псевдографики

(рис. 4.12).



```
[ Register Values Unavailable ]

B+> 0x4010e0 <_start> mov $0x4,%eax
0x4010e5 <_start+5> mov $0x1,%ebx
0x4010ea <_start+10> mov $0x402118,%ecx
0x4010ef <_start+15> mov $0x8,%edx
0x4010f4 <_start+20> int $0x80
0x4010f6 <_start+22> mov $0x4,%eax
0x4010fb <_start+27> mov $0x1,%ebx
0x401100 <_start+32> mov $0x402120,%ecx
0x401105 <_start+37> mov $0x7,%edx
0x40110a <_start+42> int $0x80

native process 4607 In: _start L9 PC: 0x4010e0
(gdb) layout regs
(gdb)
```

Рис. 4.12: Режим псевдографики

Проверяю точки останова. (рис. 4.13).

```
(gdb) info breakpoints
Num   Type             Disp Enb Address      What
1     breakpoint      keep y   0x004010e0 lab09-2.asm:9
(gdb) breakpoint already hit 1 time
(gdb)
```

Рис. 4.13: Проверка точек останова

Устанавливаю точку останова в последней инструкции. (рис. 4.14).

```
(gdb) b * 0x401111
Breakpoint 2 at 0x401111: file lab09-2.asm, line 20.
(gdb)
```

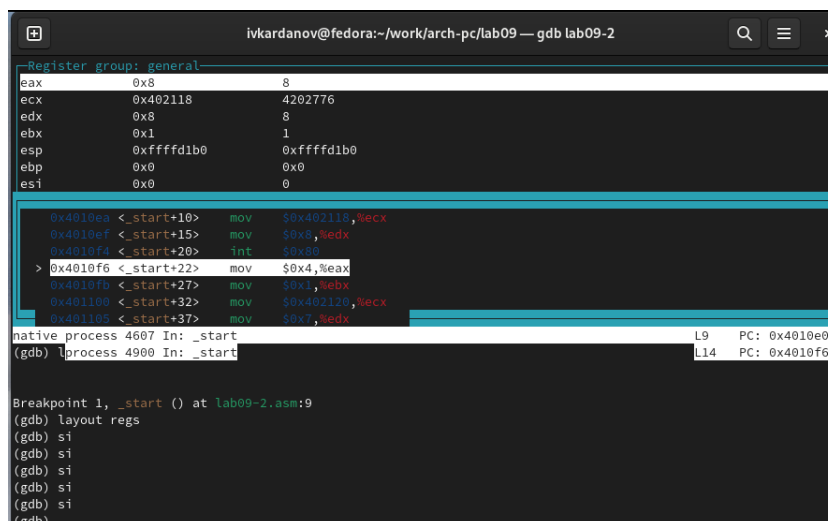
Рис. 4.14: Редактирование файла

Опять же, смотрю информацию обо всех установленных точках останова. (рис. 4.15).

```
(gdb) ↑ b
Num   Type             Disp Enb Address      What
1     breakpoint      keep y   0x004010e0 lab09-2.asm:9
      breakpoint already hit 1 time
2     breakpoint      keep y   0x00401111 lab09-2.asm:20
(gdb)
```

Рис. 4.15: Создание и запуск исполняемого файла

Вручную изменяю значений регистров и переменных с помощью инструкции si. (рис. 4.16).



```
ivkardanov@fedora:~/work/arch-pc/lab09 — gdb lab09-2
Register group: general
eax      0x8          8
ecx      0x402118    4202776
edx      0x8          8
ebx      0x1          1
esp      0xffffd1b0   0xffffd1b0
ebp      0x0          0
esi      0x0          0

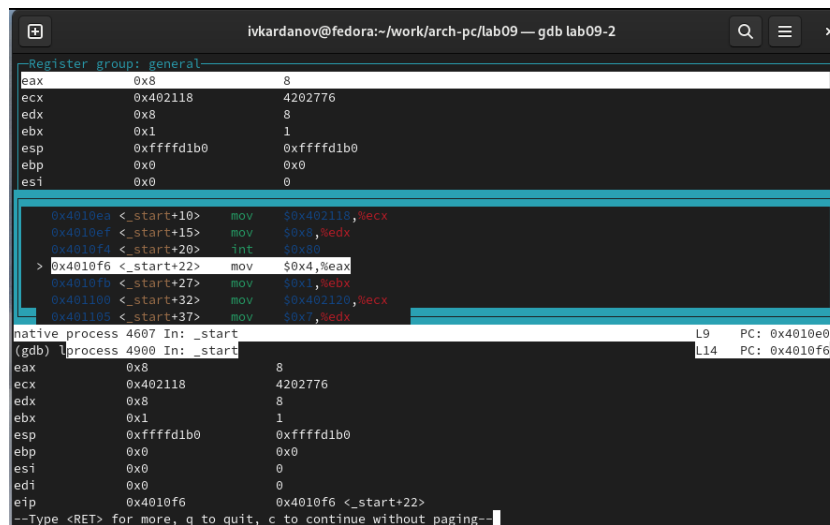
0x4010e0 <_start+10> mov $0x402118,%ecx
0x4010ef <_start+15> mov $0x8,%edx
0x4010f4 <_start+20> int $0x80
> 0x4010f6 <_start+22> mov $0x4,%eax
0x4010fb <_start+27> mov $0x1,%ebx
0x401100 <_start+32> mov $0x402120,%ecx
0x401105 <_start+37> mov $0x7,%edx

native process 4607 In: _start L9 PC: 0x4010e0
(gdb) process 4900 In: _start L14 PC: 0x4010f6

Breakpoint 1, _start () at lab09-2.asm:9
(gdb) layout regs
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb) si
(gdb)
```

Рис. 4.16: Изменение значений регистров и переменных

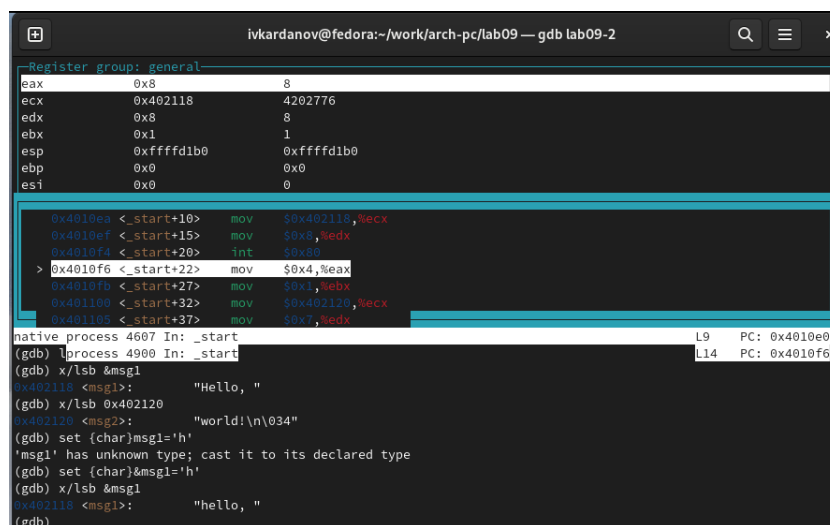
Выполняю 5 инструкций `si`, и последовательно замечаю изменение значений регистров на экране соответственно. (рис. 4.17).



The screenshot shows the GDB interface with the 'Register group: general' window at the top. The registers and their values are: `eax` (0x8, 8), `ecx` (0x402118, 4202776), `edx` (0x8, 8), `ebx` (0x1, 1), `esp` (0xffffd1b0, 0xffffd1b0), `ebp` (0x0, 0x0), and `esi` (0x0, 0). Below the register window, the assembly window shows instructions at addresses 0x4010ea to 0x401105. The instruction at 0x4010f6 is highlighted: `mov $0x4,%eax`. The console window shows the command `(gdb) process 4900 In: _start` and the current state of the registers, matching the top window. The PC is 0x4010f6.

Рис. 4.17: Изменение значений регистров и переменных

Просматриваю содержимое регистров. (рис. 4.18).



The screenshot shows the GDB interface. The 'Register group: general' window is at the top, showing the same register values as in Figure 4.17. The assembly window shows the same instructions. The console window shows the command `(gdb) x/lsb &msg1` and the output `"Hello, "`. The command `(gdb) x/lsb 0x402120` is also shown, with output `"world!\n\034"`. The command `(gdb) set {char}msg1='h'` is shown, with a warning that `'msg1' has unknown type; cast it to its declared type`. The command `(gdb) set {char}&msg1='h'` is shown, with output `"hello, "`.

Рис. 4.18: Содержимое регистров

Затем я просматриваю содержимое переменной `msg1` и изменяю в ней символ с помощью команды `{char}`. (рис. 4.19).

```

ivkardanov@fedora:~/work/arch-pc/lab09 — gdb lab09-2
Register group: general
eax      0x8      8
ecx      0x402118  4202776
edx      0x8      8
ebx      0x1      1
esp      0xffffd1b0 0xffffd1b0
ebp      0x0      0x0
esi      0x0      0

0x4010ea <_start+10> mov $0x402118,%ecx
0x4010ef <_start+15> mov $0x8,%edx
0x4010f4 <_start+20> int $0x80
> 0x4010f6 <_start+22> mov $0x4,%eax
0x4010fb <_start+27> mov $0x1,%ebx
0x401100 <_start+32> mov $0x402120,%ecx
0x401105 <_start+37> mov $0x7,%edx

native process 4607 In: _start
(gdb) !process 4900 In: _start
0x402100 <msg2>: "world!\n\034"
(gdb) set {char}msg1='h'
'msg1' has unknown type; cast it to its declared type
(gdb) set {char}&msg1='h'
(gdb) x/lsb &msg1
0x402118 <msg1>: "hello, "
(gdb) set {char}0x402120 ='E'
(gdb) x/lsb 0x402120
0x402120 <msg2>: "Eorld!\n\034"
(gdb)

```

Рис. 4.19: Содержимое переменной, изменение в ней символов

Аналогичные действия проделываю с переменной msg2. (рис. 4.20).

```

ivkardanov@fedora:~/work/arch-pc/lab09 — gdb lab09-2
Register group: general
eax      0x8      8
ecx      0x402118  4202776
edx      0x8      8
ebx      0x1      1
esp      0xffffd1b0 0xffffd1b0
ebp      0x0      0x0
esi      0x0      0x0

0x4010ea <_start+10> mov $0x402118,%ecx
0x4010ef <_start+15> mov $0x8,%edx
0x4010f4 <_start+20> int $0x80
> 0x4010f6 <_start+22> mov $0x4,%eax
0x4010fb <_start+27> mov $0x1,%ebx
0x401100 <_start+32> mov $0x402120,%ecx
0x401105 <_start+37> mov $0x7,%edx

native process 4607 In: _start
(gdb) !process 4900 In: _start
0x402100 <msg2>: "world!\n\034"
(gdb) set {char}0x402120 ='E'
(gdb) x/lsb 0x402120
0x402120 <msg2>: "Eorld!\n\034"
(gdb) p/s $edx
$1 = 8
(gdb) p/t $edx
$2 = 1000
(gdb) p/x $edx
$3 = 0x8
(gdb)

```

Рис. 4.20: Содержимое переменной, изменение в ней символов

Ввожу в различных форматах значение регистра edx. (рис. 4.21).

```

ivkardanov@fedora:~/work/arch-pc/lab09 — gdb lab09-2
Register group: general
eax      0x8          8
ecx      0x402118     4202776
edx      0x8          8
ebx      0x2          2
esp      0xffffd1b0   0xffffd1b0
ebp      0x0          0
esi      0x0          0

0x4010e9 <_start+10> mov     $0x402118,%ecx
0x4010ef <_start+15> mov     $0x8,%edx
0x4010f4 <_start+20> int     $0x80
> 0x4010f6 <_start+22> mov     $0x4,%eax
0x4010f6 <_start+27> mov     $0x1,%ebx
0x401100 <_start+32> mov     $0x402120,%ecx
0x401105 <_start+37> mov     $0x7,%edx

native process 4607 In: _start
(gdb) l process 4900 In: _start
$2 = 1000
(gdb) p/x $edx
$3 = 0x8
(gdb) set $ebx = '2'
(gdb) p/s $ebx
$4 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$5 = 2
(gdb)

```

Рис. 4.21: Значения регистра edx

Изменяю значение регистра ebx с помощью команды set. (рис. 4.22).

```

[ivkardanov@fedora lab09]$ cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm
[ivkardanov@fedora lab09]$ nasm -f elf -g -l lab09-3.lst lab09-3.asm
[ivkardanov@fedora lab09]$ ld -m elf_i386 -o lab09-3 lab09-3.o
[ivkardanov@fedora lab09]$

```

Рис. 4.22: Изменение значений регистра

Разница в выводе команд объясняется в значении: при бескавычном значении 2, мы её и получаем в итоге, а в другом случае переменная воспринимается иначе, и на выходе мы видим значение 50.

Завершаю выполнение программы с помощью continue и выхожу из gdb с помощью quit.

Копирую файл lab8-2.asm, полученный во время выполнения лабораторной работы №8, содержащий программу для вывода аргументов командной строки. Затем создаю исполняемый файл. (рис. 4.23).

```
[ivkardanov@fedora lab09]$ gdb --args lab09-3 argument1 argument 2 'argument 3'
GNU gdb (GDB) Fedora Linux 13.2-6.fc38
Copyright (C) 2023 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 lab09-3...
(gdb)
```

Рис. 4.23: Копирование файла, создание исполняемого файла

Загружаю исполняемый файл в отладчик, указав нужные аргументы. (рис. 4.24).

```
(gdb) b _start
Breakpoint 1 at 0x4011a8: file lab09-3.asm, line 5.
(gdb) run
Starting program: /home/ivkardanov/work/arch-pc/lab09/lab09-3 argument1 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]) n
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.

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

Рис. 4.24: Загрузка файла в отладчик

Устанавливаю точку останова перед первой инструкцией и запускаю программу. (рис. 4.25).

```
(gdb) x/x $esp
0xffffd180: 0x00000005
(gdb) x/s *(void**)(esp + 4)
0xffffd333: "/home/ivkardanov/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd35f: "argument1"
(gdb) x/s *(void**)(esp + 12)
0xffffd369: "argument"
(gdb) x/s *(void**)(esp + 16)
0xffffd372: "2"
(gdb) x/s *(void**)(esp + 20)
0xffffd374: "argument 3"
(gdb) x/s *(void**)(esp + 24)
0x0: <error: Cannot access memory at address 0x0>
(gdb)
```

Рис. 4.25: Установка точки останова, запуск программы

Далее просматриваю позиции стека. (рис. 4.26).

```
[ivkardanov@fedora lab09]$ cp ~/work/arch-pc/lab08/ik.asm ~/work/arch-pc/lab09/ik-1.asm
[ivkardanov@fedora lab09]$ gedit ik-1.asm

ik-1.asm
~/work/arch-pc/lab09

1 %include 'in_out.asm'
2 SECTION .data
3 msg db "Результат: ",0
4 a: db 'f(x) = 3(x+2)',0
5 SECTION .text
6 global _start
7 _start:
8 mov eax, a
9 call sprintf
10 pop ecx ; Извлекаем из стека в 'ecx' количество
11 ; аргументов (первое значение в стеке)
12 pop edx ; Извлекаем из стека в 'edx' имя программы
13 ; (второе значение в стеке)
14 sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
15 ; аргументов без названия программы)
16 mov esi, 0 ; Используем 'esi' для хранения
17 ; промежуточных сумм
18 next:
19 cmp ecx,0h ; проверяем, есть ли еще аргументы
20 jz _end ; если аргументов нет выходим из цикла
21 ; (переход на метку '_end')
22 pop eax ; иначе извлекаем следующий аргумент из стека
23 call atoi ; преобразуем символ в число
24 call _calc
25 add esi,eax ; добавляем к промежуточной сумме
26 loop next ; переход к обработке следующего аргумента
27 _end:
28 mov eax, msg ; вывод сообщения "Результат: "
29 call sprintf
30 mov eax, esi ; записываем сумму в регистр 'eax'
31 call iprintf ; печать результата
32 call quit ; завершение программы
33 _calc:
34 mov esi, 0
```

Рис. 4.26: Просмотр позиций стека

Шаг изменения равен 4, т.к. каждый следующий адрес на стеке находится на расстоянии в 4 байта от предыдущего.

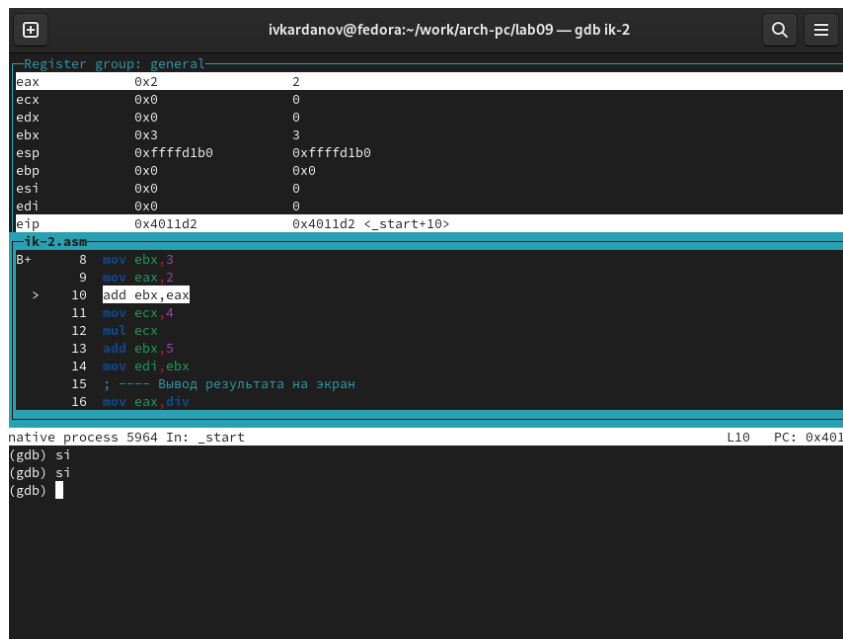
4.3) Выполнение заданий для самостоятельной работы

Копирую файл задания для самостоятельной работы, и реализую вычисление значения функции через подпрограмму. (рис. 4.27).

```
[ivkardanov@fedora lab09]$ ld -m elf_i386 ik-1.o -o ik-1
[ivkardanov@fedora lab09]$ ./ik-1 1
f(x) = 3(x+2)
Результат: 5
[ivkardanov@fedora lab09]$ ./ik-1 1 4 6 8
f(x) = 3(x+2)
Результат: 140
[ivkardanov@fedora lab09]$
```

Рис. 4.27: Копирование, редактирование файла

Создаю исполняемый файл и убеждаюсь в правильности работы программы. (рис. 4.28).



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

Register	Value
eax	0x2
ecx	0x0
edx	0x0
ebx	0x3
esp	0xffffd1b0
ebp	0x0
esi	0x0
edi	0x0
eip	0x4011d2

Below the registers, the assembly code for "ik-2.asm" is shown:

```
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
```

The bottom of the window shows the GDB prompt with the command "si" entered, and the status bar indicates "native process 5964 In: _start" and "PC: 0x4011d2".

Рис. 4.28: Создание и запуск исполняемого файла

Создаю файл `ik-2.asm` и вношу в него программу из последнего листинга. При запуске программа дает неверный результат, и дабы исправить эту ситуацию, нужно проанализировать изменения значений регистров, что я и сделал. Благодаря этому мне удалось вычислить ошибку и исправить её в тексте программы. (рис. ??).

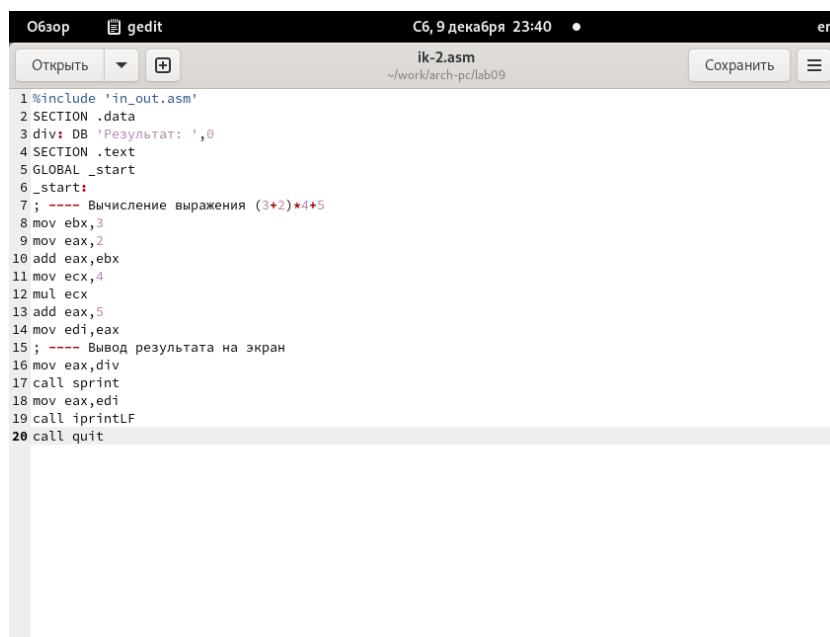


Рис. 4.29: Редактирование файла

Создаю исполняемый файл, и, выполнив устную проверку, убеждаемся в правильности работы программы. (рис. 4.29).

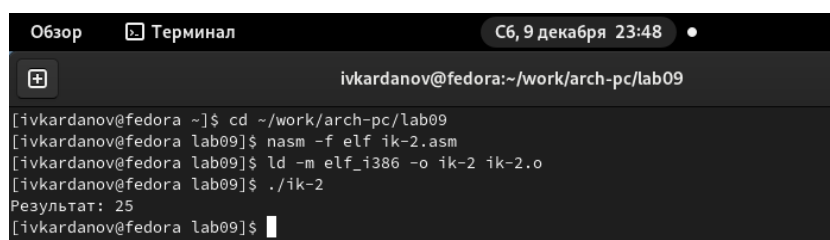


Рис. 4.30: Создание и запуск исполняемого файла

Листинг 4.1 - Преобразованная программа из лабораторной работы №8.

```

““%include 'in_out.asm'
SECTION .data
msg db “Результат:”,0
a: db ‘f(x) = 3(x+2)’,0
SECTION .text

```

```

global _start
_start:
mov eax, a
call sprintLF
pop ecx ; Извлекаем из стека в ecx количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в edx имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем ecx на 1 (количество
; аргументов без названия программы)
mov esi, 0 ; Используем esi для хранения
; промежуточных сумм
next:
cmp ecx,0h ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку _end)
pop eax ; иначе извлекаем следующий аргумент из стека
call atoi ; преобразуем символ в число
call _calc
add esi,eax ; добавляем к промежуточной сумме
loop next ; переход к обработке следующего аргумента
_end:
mov eax, msg ; вывод сообщения "Результат:"
call sprint
mov eax, esi ; записываем сумму в регистр eax
call iprintLF ; печать результата
call quit ; завершение программы
_calc:
imul eax,8

```



```
add eax,-3
ret
```

Листинг 4.2 - Исправленная программа для вычисления значения выражения.

```
` ``%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 eax,ebx

mov ecx,4
```

```
mul ecx
```

```
add eax,5
```

```
mov edi,eax
```

```
; ---- Вывод результата на экран
```

```
mov eax,div
```

```
call sprint
```

```
mov eax,edi
```

```
call iprintLF
```

```
call quit
```

5 Выводы

При выполнении лабораторной работы я приобрел практический опыт в написании программ в написании программ с использованием подпрограмм, а также ознакомился с методами отладки при помощи gdb и его основными возможностями.

Список литературы

Архитектура компьютера и ЭВМ