Понятие подпрограммы. Отладчик GDB

Лабораторная работа №9

Приходько Иван Иванович

Содержание

Список иллюстраций

Список таблиц

1 Цель работы

Ознакомиться с понятием подпрограмм в Ассемблере и научиться использовать подпрограммы на практике. Ознакомиться с отладчиком gdb и научиться использовать его

2 Выполнение лабораторной работы

Для начала выполнения работы необходимо создать файл lab9-1.asm (рис. 2.1).

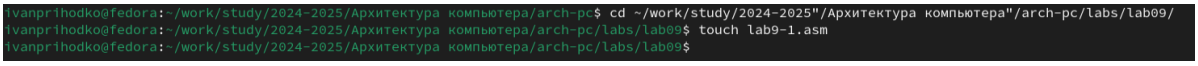

Рис. 2.1: Создание lab9-1.asm

Скопируем файл in_out.asm из директории прошлой работы (рис. 2.2).

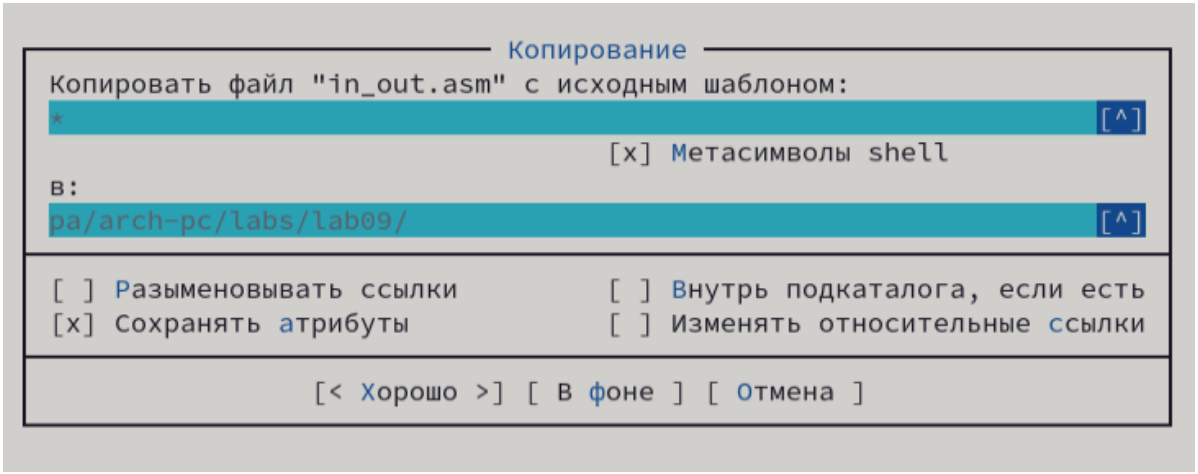

Рис. 2.2: Копирование in_out.asm

Вставим в файл lab9-1.asm код из листинга 9.1 (рис. 2.3).

```
  GNU nano 7.2                                                /
%include 'in_out.asm'
SECTION .data
msg: DB 'Введите x: ',0
result: DB '2x+7=',0
SECTION .bss
x: RESB 80
res: 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,[res]
call iprintLF
call quit
;-----------------------------------------
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret ; выход из подпрограммы
```

Рис. 2.3: Вставка кода из файла листинга 9.1

Соберем и запустим файл (рис. 2.4).

```
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ nasm -f elf lab9-1.asm
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ./lab9-1
Введите x: 10
2x+7=27
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$
```

Рис. 2.4: Сборка и запуск lab9-1.asm

Теперь изменим файл так, чтобы внутри подпрограммы была ещё одна подпрограмма, вычисляющая значение g(x) и чтобы она передавала значение в первую подпрограмму, которая бы уже вычислила значение f(g(x)) (рис. 2.5).

```
call quit
;----------------------------------
; Подпрограмма вычисления
; выражения "2x+7"
_calcul:
call _subcalcul
mov ebx,2
mul ebx
add eax,7
mov [res],eax
_subcalcul:
mov ebx,3
mul ebx
sub eax,1
ret

ret ; выход из подпрограммы
```

Рис. 2.5: Изменение файла lab9-1.asm

Повторно соберем и запустим программу (рис. 2.6).

```
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ nasm -f elf lab9-1.asm
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ./lab9-1
Введите x: 1
f(g(x))=11
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ./lab9-1
Введите x: 2
f(g(x))=17
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ./lab9-1
Введите x: 3
f(g(x))=23
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$
```

Рис. 2.6: Повторная сборка и запуск lab9-1.asm

Создадим новый файл и вставим в него код из листинга 9.2 (рис. 2.7).

```
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ touch lab9-2.asm
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$
```

Рис. 2.7: Создание lab9-2.asm

```
 GNU nano 7.2
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. 2.8: Редактирование lab9-2.asm

Соберём программу с использованием аргумента -g (рис. 2.9).

```
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ nasm -f elf -g -l lab9-2.lst lab9-2.asm
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ ld -m elf_i386 -o lab9-2 lab9-2.o
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$
```

Рис. 2.9: Сборка lab9-2.asm

Теперь загрузим её в gdb (рис. 2.10).

```
ivanprihodko@fedora:~/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09$ gdb lab9-2
GNU gdb (Fedora Linux) 14.2-1.fc40
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 lab9-2...
(gdb)
```

Рис. 2.10: Загрузка программы lab9-2.asm в gdb

Создадим брейкпоинт на метке _start с помощью команды break (рис. 2.11).

```
(gdb) break _start
Breakpoint 1 at 0x8049000: file lab9-2.asm, line 9.
(gdb) run
Starting program: /home/ivanprihodko/work/study/2024-2025/Архитектура компьютера/arch-pc/labs/lab09/lab9-2

Breakpoint 1, _start () at lab9-2.asm:9
9       mov eax, 4
(gdb)
```

Рис. 2.11: Создание брейкпоинта

С помощью команды disassemble дизассемблируем её (рис. 2.12).

Рис. 2.12: Дизассемблирование программы

Переключим синтаксис вывода на intel и повторно дизассемблируем программу (рис. 2.13).

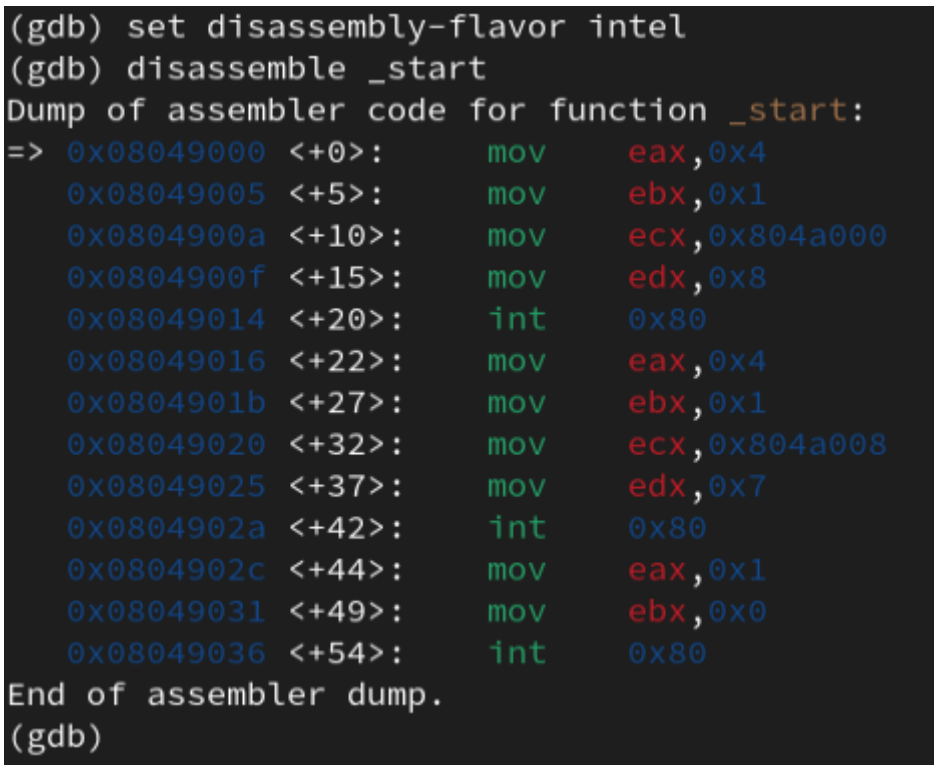

Рис. 2.13: Переключение на синтаксис intel

Включим графическое отображения кода (рис. 2.14).

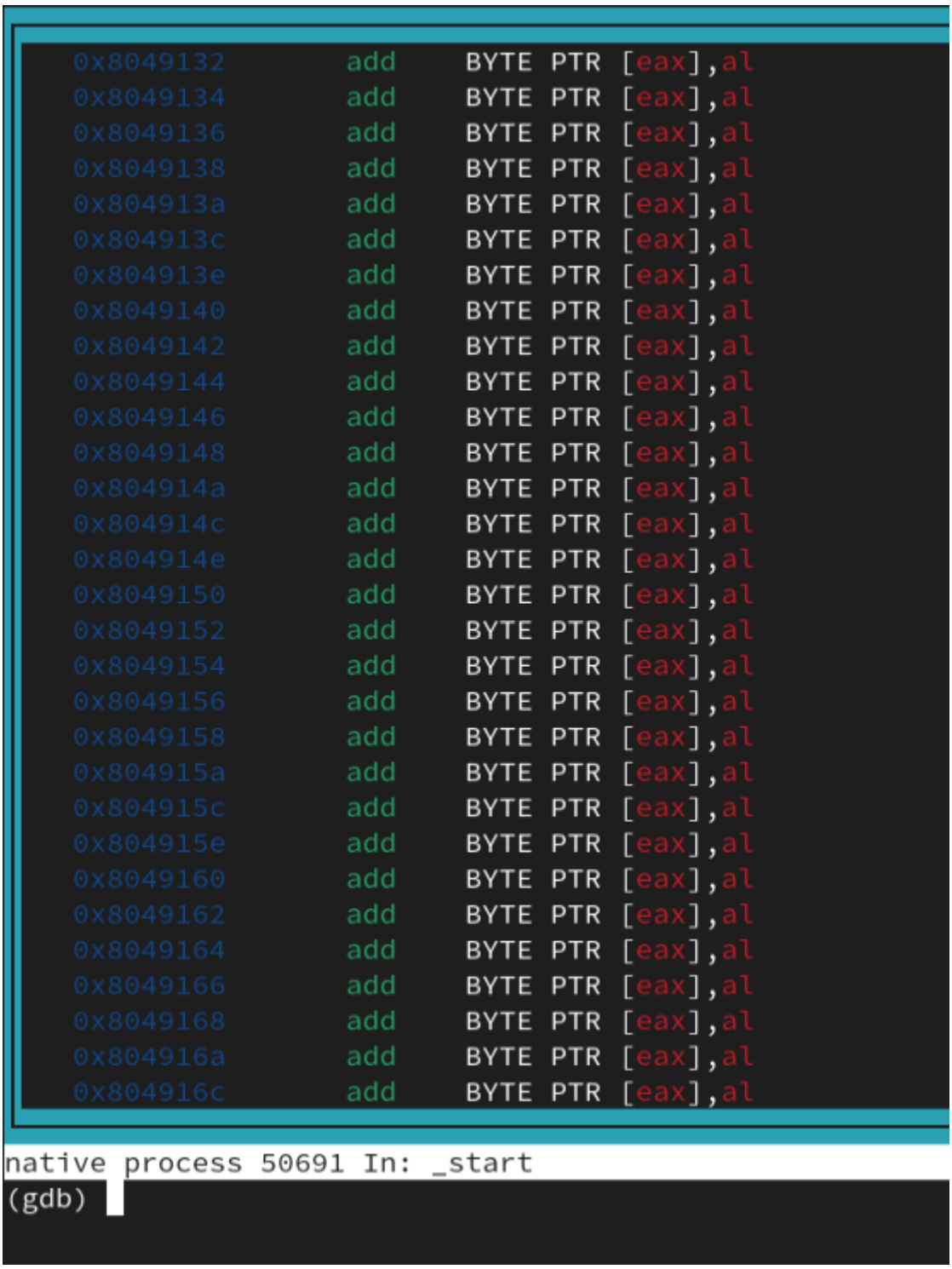

Рис. 2.14: Внешний вид интерфейса

Включим графическое отображение значений регистров (рис. 2.15).

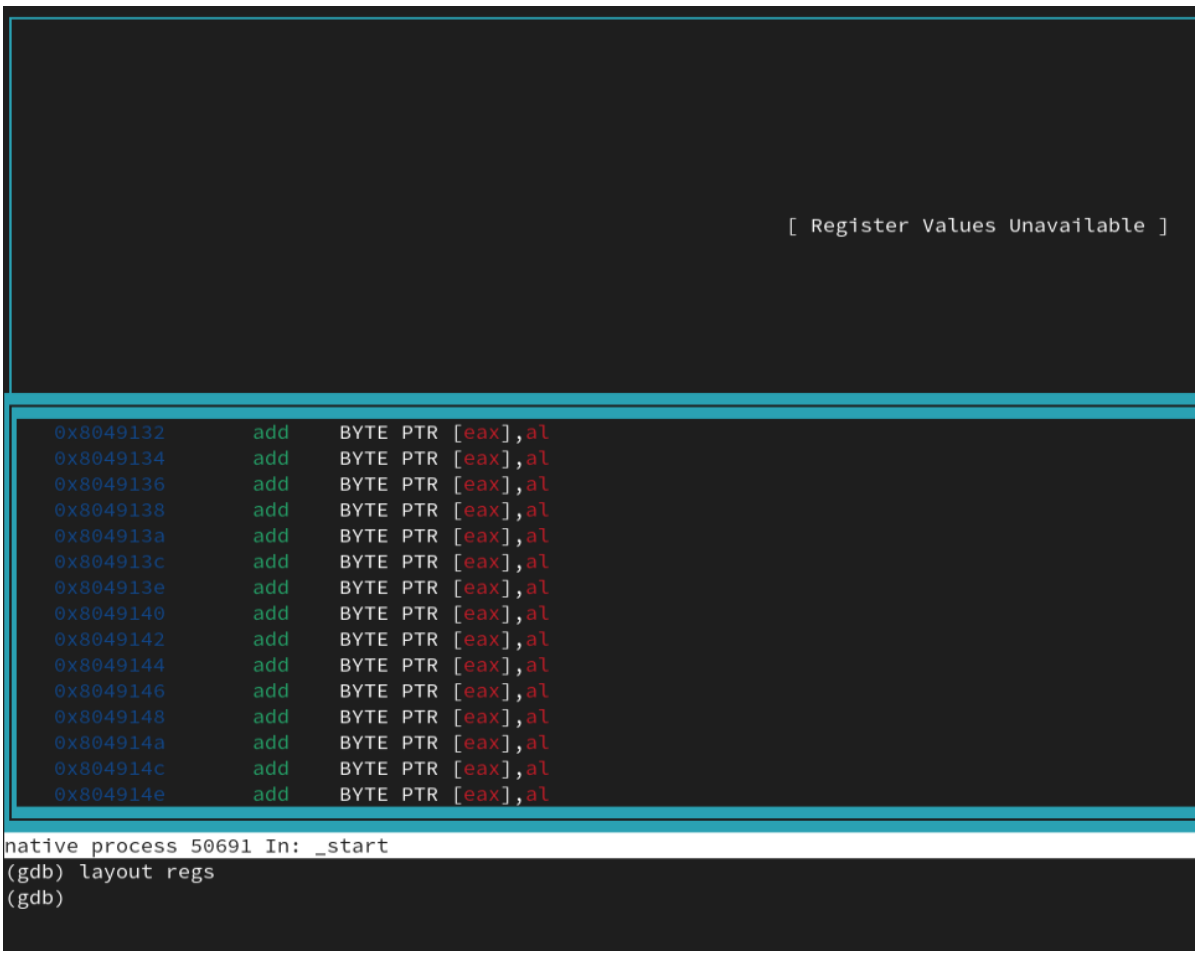

Рис. 2.15: Включение графического отображения значений регистров

Выведем информацию о всех брейкпоинтах (рис. 2.16).

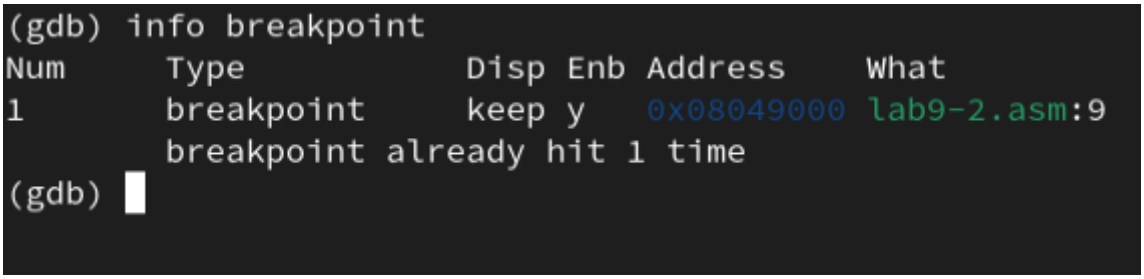

Рис. 2.16: Вывод информации о брейкпоинтах

Попробуем теперь создать брейкпоинт по адресу (рис. 2.17).

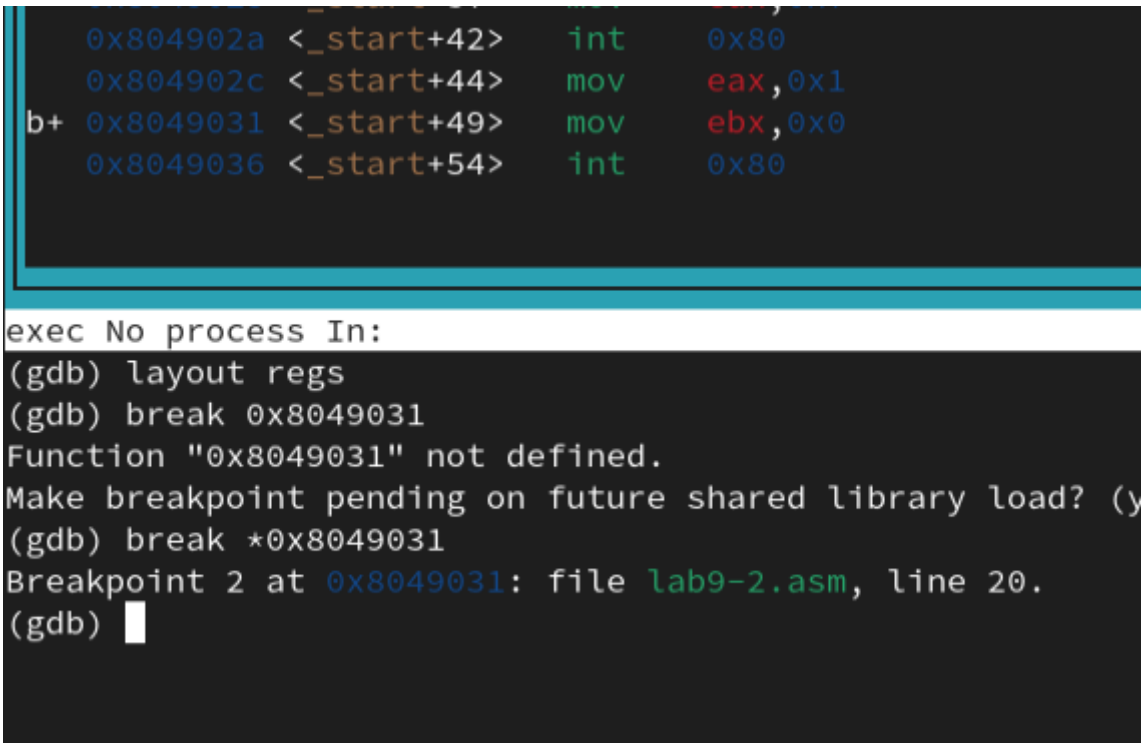

Рис. 2.17: Создание брейкпоинта по адресу

Повторно выведем информацию о брейкпоинтах (рис. 2.18).

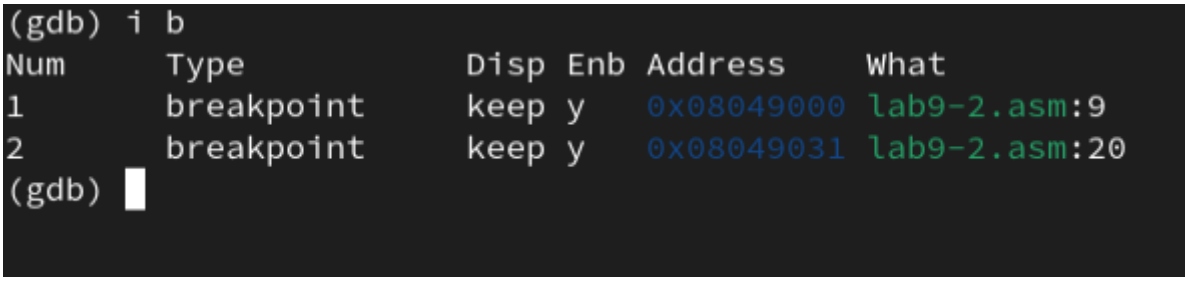

Рис. 2.18: Повторный вывод информации о брейкпоинтах

Теперь 5 раз выполним команду si для построчного выполнения кода (рис. 2.19-2.23).

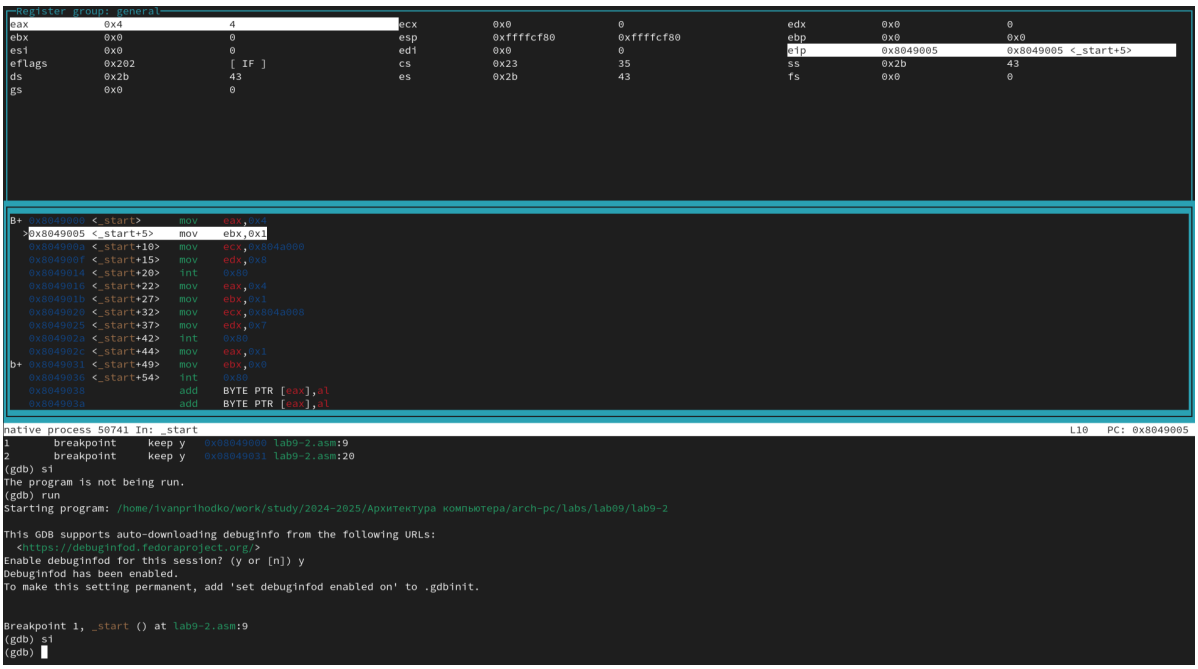

Рис. 2.19: Выполнение команды в коде программы (1)

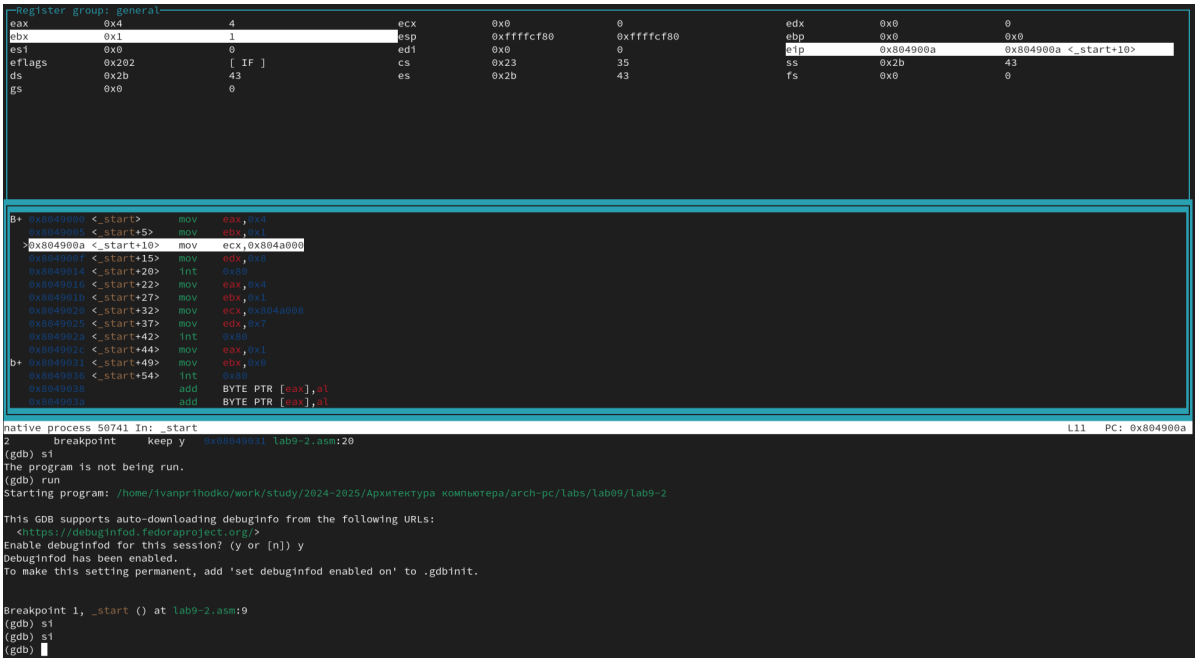

Рис. 2.20: Выполнение команды в коде программы (2)

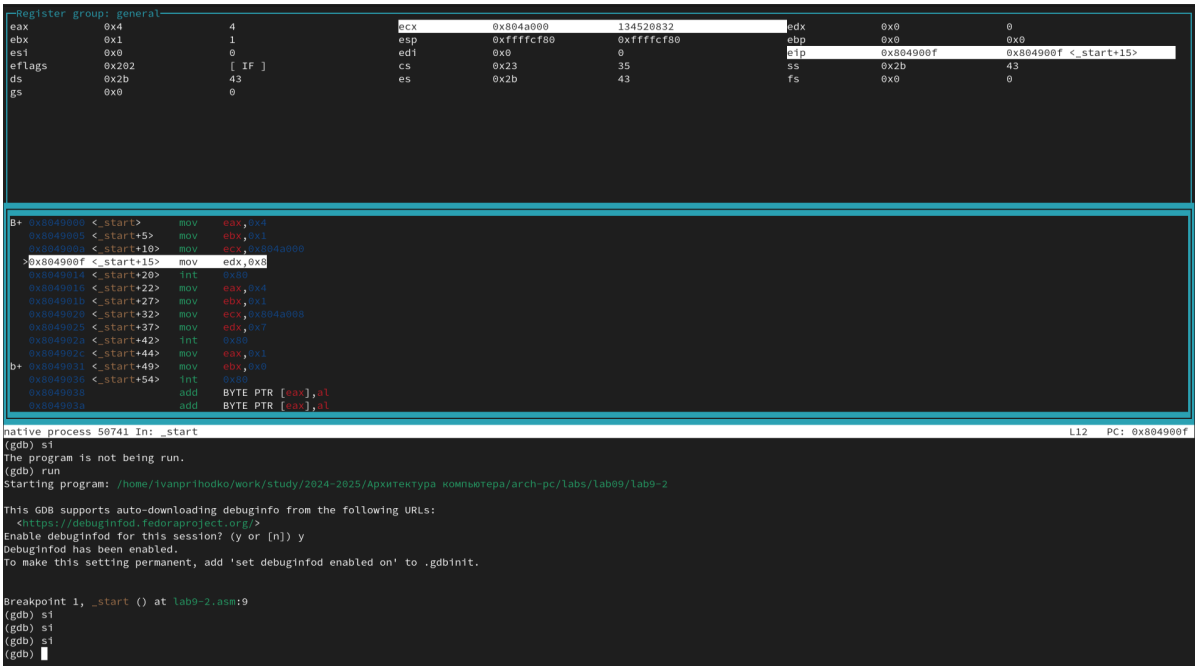

Рис. 2.21: Выполнение команды в коде программы (3)

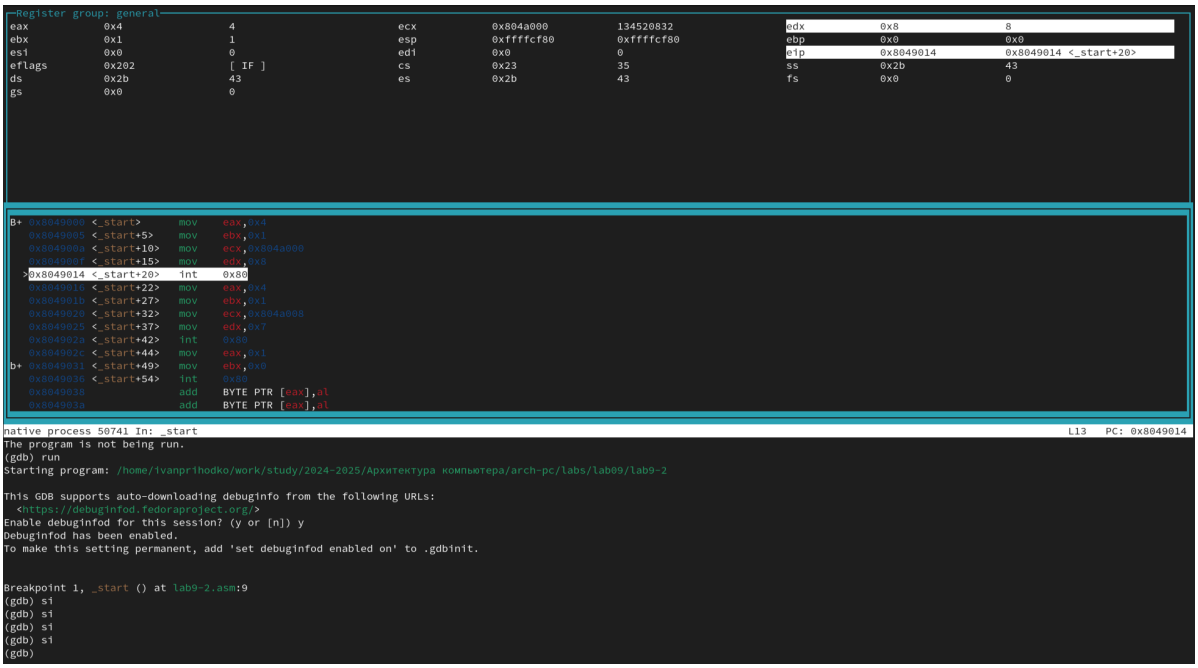

Рис. 2.22: Выполнение команды в коде программы (4)

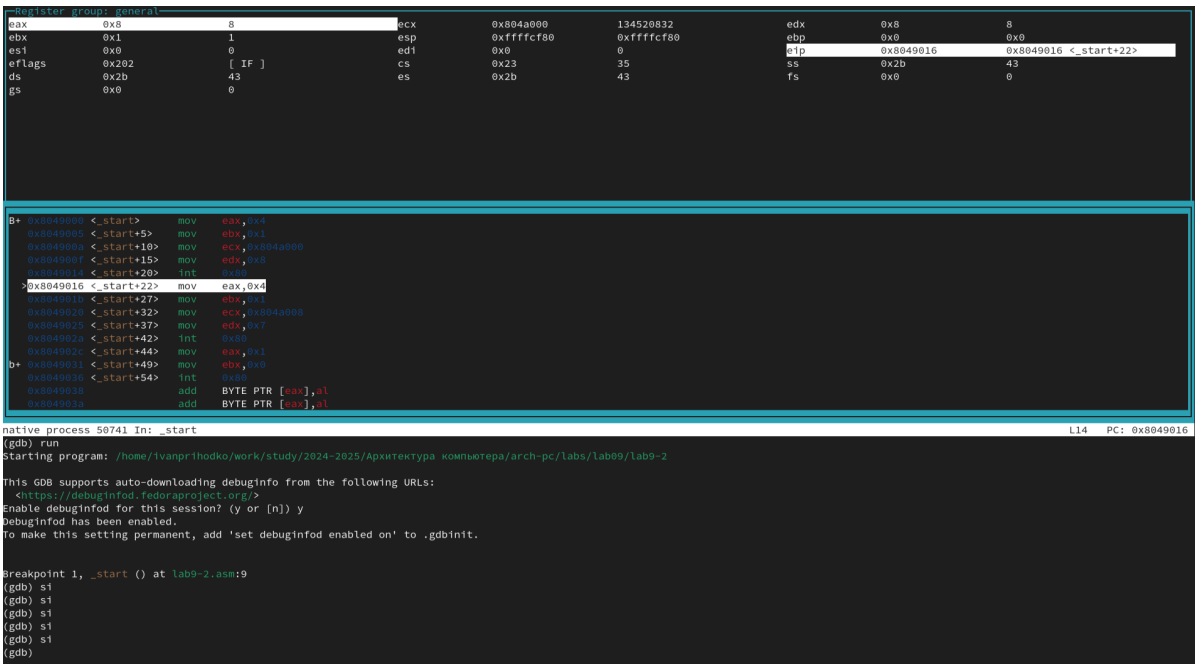

Рис. 2.23: Выполнение команды в коде программы (5)

Как видим, поменялись значения регистров eax, ecx, edx и ebx. Теперь выведем информацию о значениях регистров (рис. 2.24).

Рис. 2.24: Вывод значений регистров

```
eax            0x8                8
ecx            0x804a000          134520832
edx            0x8                8
ebx            0x1                1
esp            0xffffcf80         0xffffcf80
ebp            0x0                0x0
esi            0x0                0
edi            0x0                0
eip            0x8049016          0x8049016 <_start+22>
eflags         0x202              [ IF ]
cs             0x23               35
ss             0x2b               43
ds             0x2b               43
es             0x2b               43
fs             0x0                0
gs             0x0                0
```

Рис. 2.25: Значения регистров

Попробуем вывести значение переменной по имени (рис. 2.26).

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

Рис. 2.26: Вывод значения переменной по имени

Теперь попробуем вывести значение переменной по адресу (рис. 2.27).

```
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:       "world!\n\034"
(gdb)
```

Рис. 2.27: Вывод значения переменной по адресу

Теперь изменим первый символ переменной (рис. 2.28).

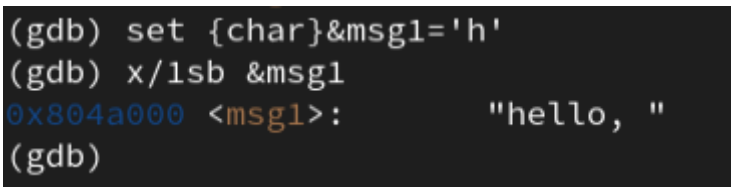

Рис. 2.28: Изменение первого символа переменной по имени и вывод переменной

А теперь изменим второй символ переменной, уже обратившись по адресу (рис. 2.29).

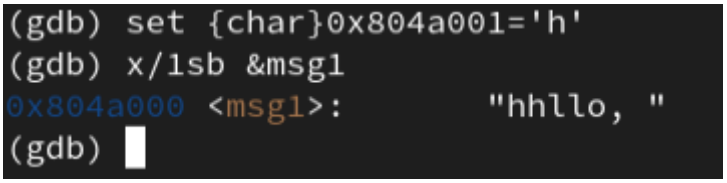

Рис. 2.29: Изменение второго символа переменной по адресу и вывод переменной

Теперь изменим несколько символов второй переменной (рис. 2.30).

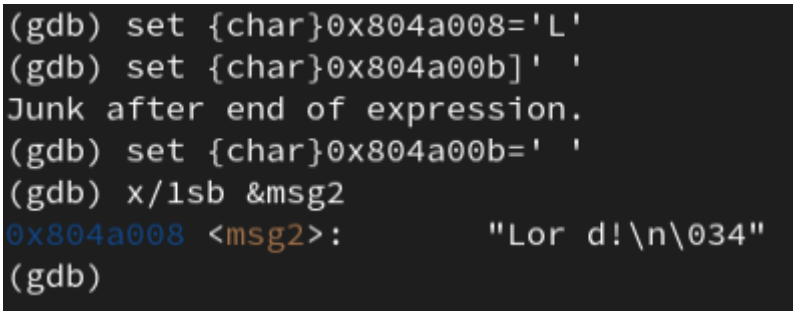

Рис. 2.30: Изменение нескольких символов второй переменной по адресу и вывод переменной

Теперь попробуем вывести значение регистра в строковом, двоичном и шестнадцатиричном виде (рис. 2.31).

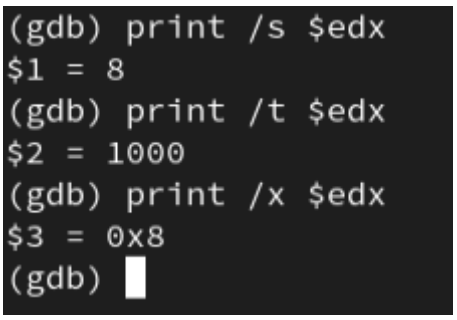

Рис. 2.31: Вывод значения регистра в строковом, двоичном и шестнадцатиричном виде

Попробуем теперь изменить значение регистра (рис. 2.32).

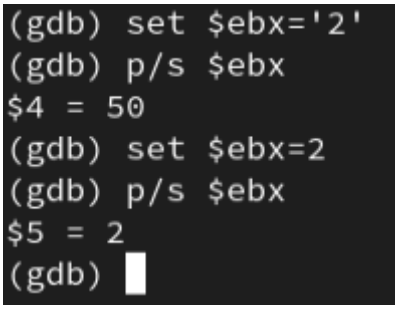

Рис. 2.32: Изменение значения регистра

Как видим, в регистр записались разные значения. Это связано с тем, что в одном случае мы записываем в него число, а в другом случае - строку. Завершим работу программы (рис. 2.33-2.34).

Рис. 2.33: Завершение работы программы

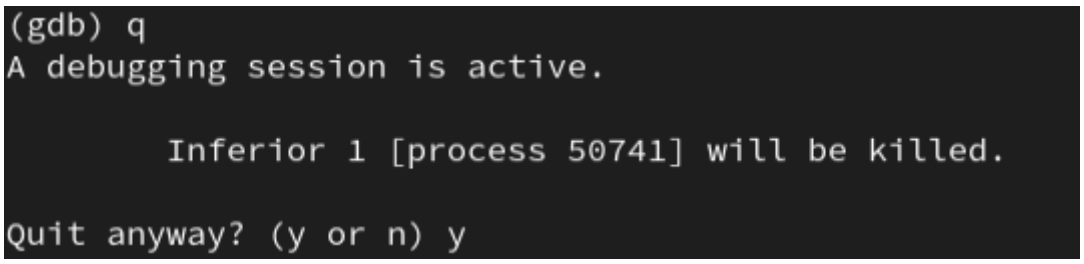

Рис. 2.34: Завершение работы программы

Скопируем файл из прошлой работы, соберём его и выгрузим в gdb (рис. 2.35).

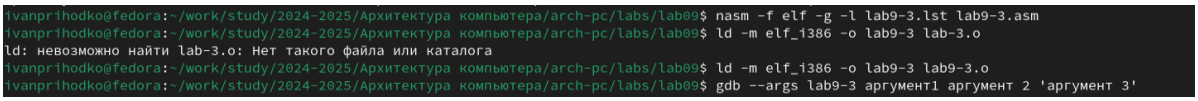

Рис. 2.35: Сборка программы и выгрузка в gdb

Создадим брейкпоинт и запустим программу (рис. 2.36).

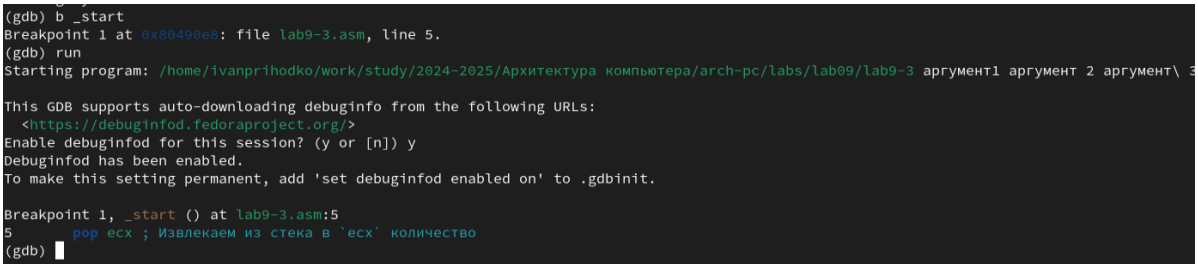

Рис. 2.36: Создание брейкпоинта и запуск программы

Теперь выведем значение регистра esp, где хранятся данные о стеке (рис. 2.37).

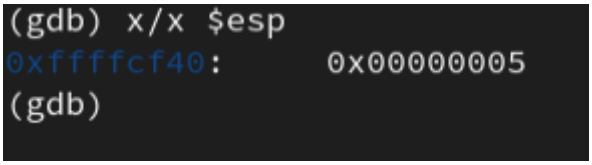

Рис. 2.37: Вывод значения регистра esp

Теперь выведем значение всех элементов стека (рис. 2.38).

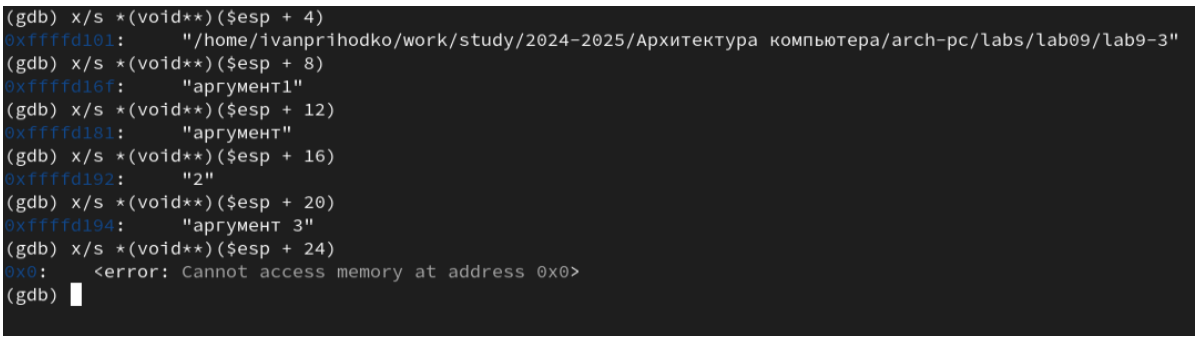

Рис. 2.38: Вывод всех значений в стеке

Как видим, для вывода каждого элемента стека нам нужно менять значение адреса с шагом 4. Это связано с тем, что именно с шагом 4 располагаются данные в стеке, ведь под каждый элемент выделяется 4 байта.

3 Выполнение задания для самостоятельной работы

Скопируем файл первого задания прошлой самостоятельной работы и перепишем его так, чтобы он использовал для вычисления выражения подпрограмму (рис. 3.1).

```
 GNU nano 7.2                                  /home/iv
%include 'in_out.asm'
SECTION .data
msg db "Результат: ",0
msg2 db "Функция: f(x)=4x-3",0
SECTION .text
global _start
_start:
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 _calcul
add esi,eax ; добавляем к промежуточной сумме
; след. аргумент `esi=esi+eax`
loop next ; переход к обработке следующего аргумента
_end:
mov eax, msg2
call sprintLF
mov eax, msg ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр `eax`
call iprintLF ; печать результата
call quit ; завершение программы
_calcul:
mov ebx, 4
mul ebx
sub eax, 3
ret
```

Рис. 3.1: Редактирование кода

Соберем и запустим (рис. 3.2).

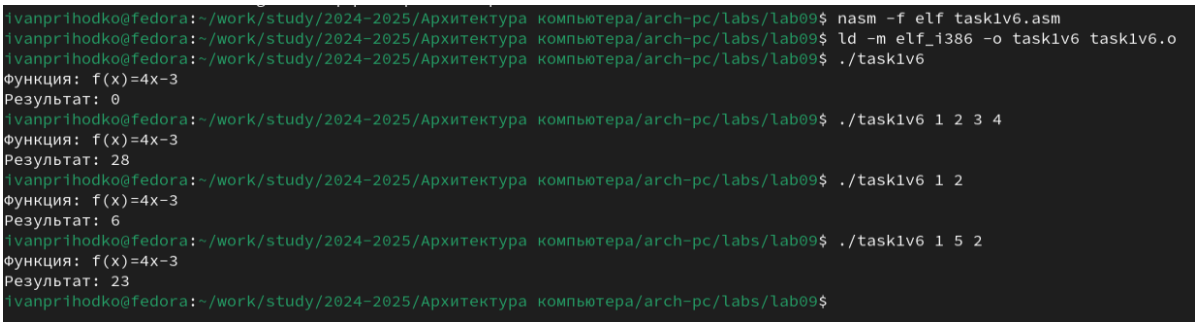

Рис. 3.2: Сборка и проверка работы программы

Создадим файл второго задания самостоятельной работы (рис. 3.3).

Рис. 3.3: Создание файла второго задания самостоятельной работы

Вставим в него код из листинга 9.3 (рис. 3.4).

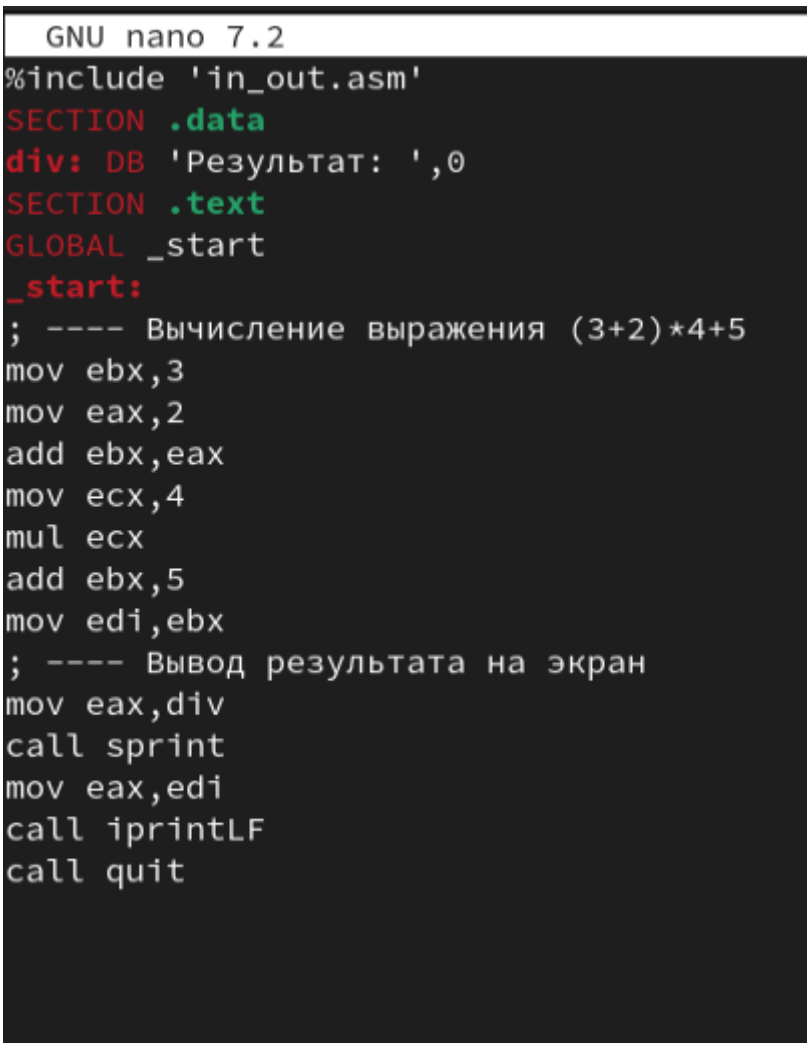

Рис. 3.4: Вставка кода из листинга 9.3

Соберем его и запустим (рис. 3.5).

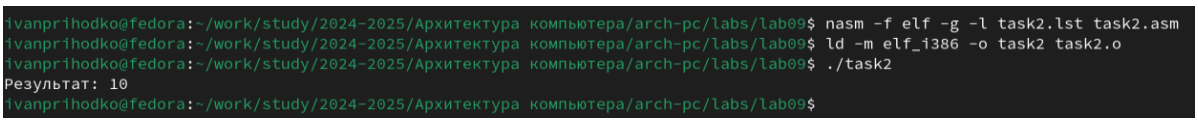

Рис. 3.5: Сборка и запуск программы

Как видим, код считает значение выражения неправильно. Загрузим его в gdb (рис. 3.6).

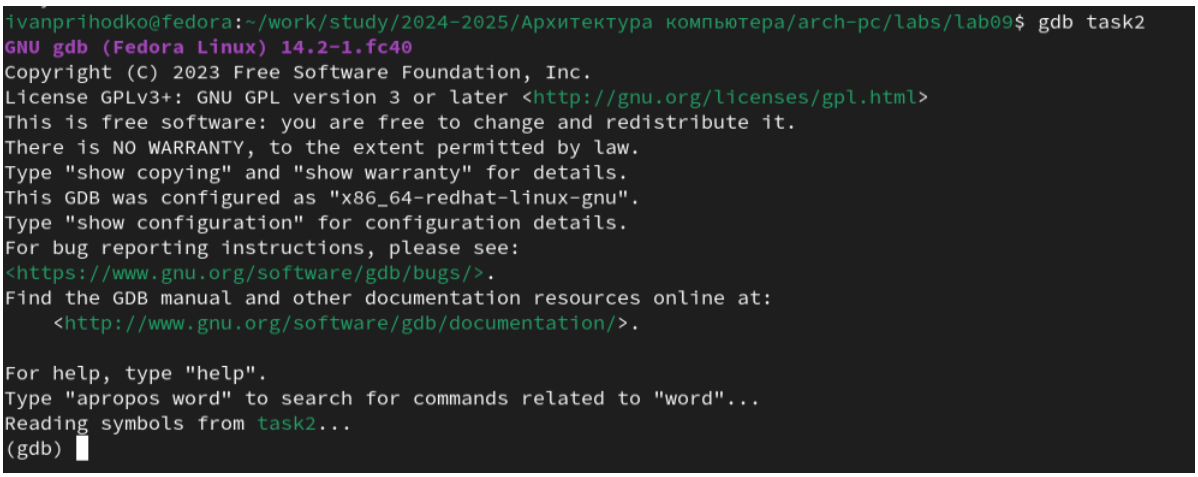

Рис. 3.6: Выгрузка программы в gdb

Переключим его на синтаксис intel (рис. 3.7).

Рис. 3.7: Переключение на синтаксис intel

Включим графическое отображение кода, отображение значений регистров и установим брейкпоинт на _start (рис. 3.8).

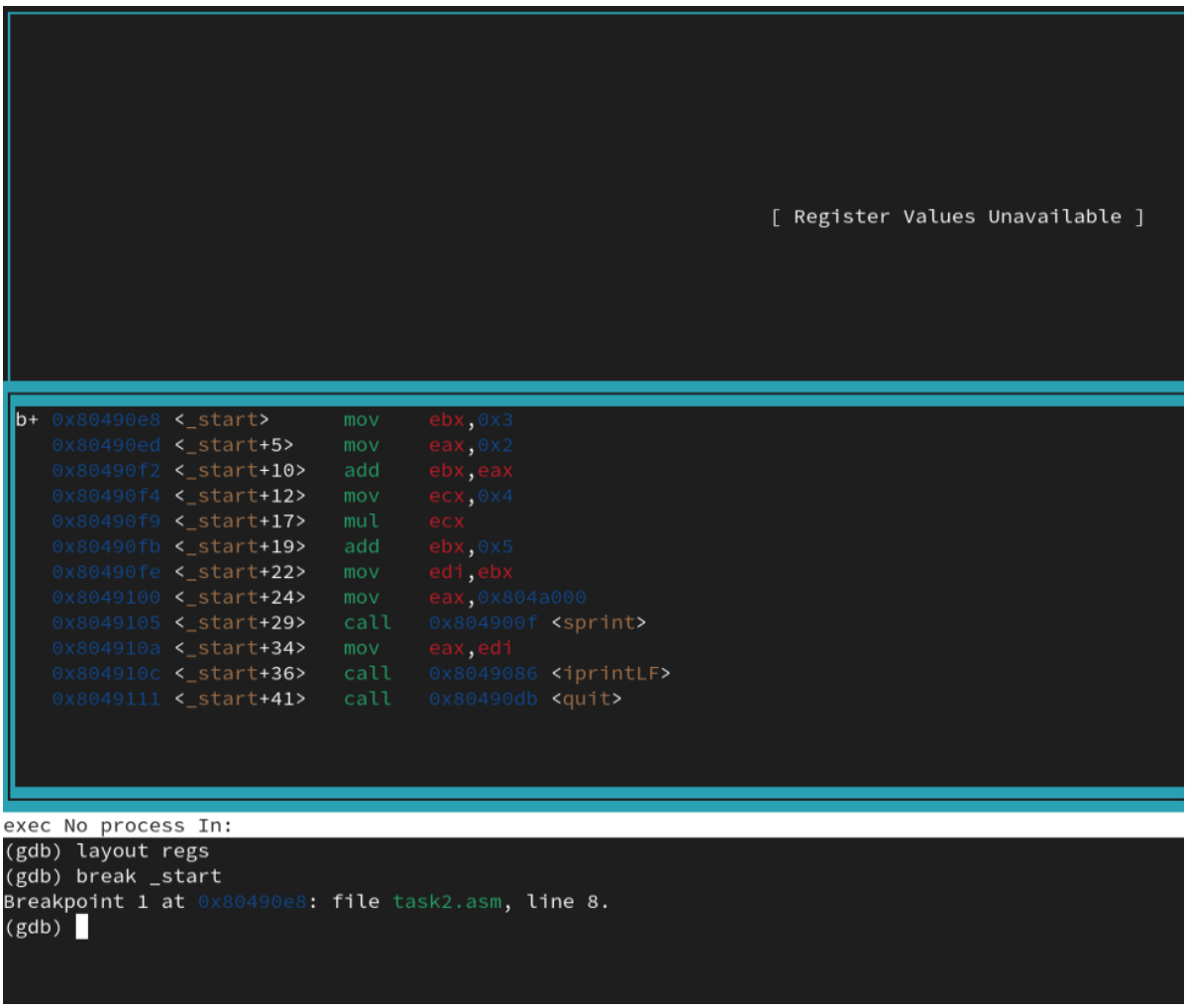

Рис. 3.8: Включение графического отображения кода и выполнения команд

И начнём построчно выполнять код (рис. 3.9 - 3.14).

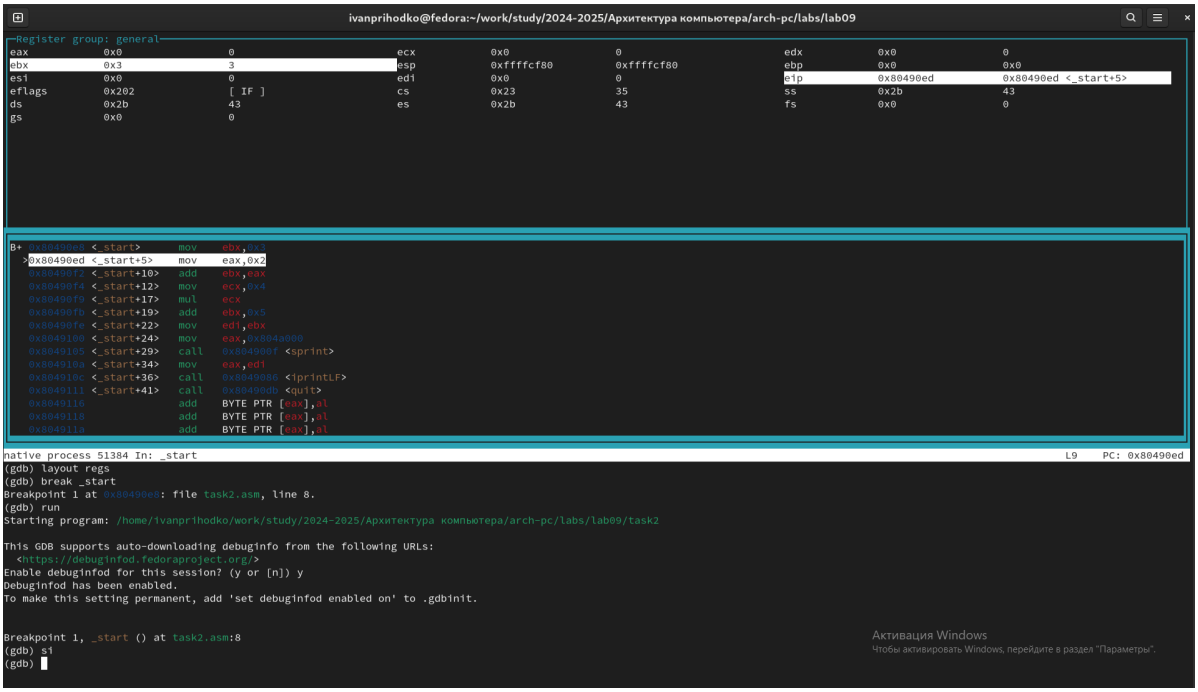

Рис. 3.9: Значение всех регистров на 1 шаге

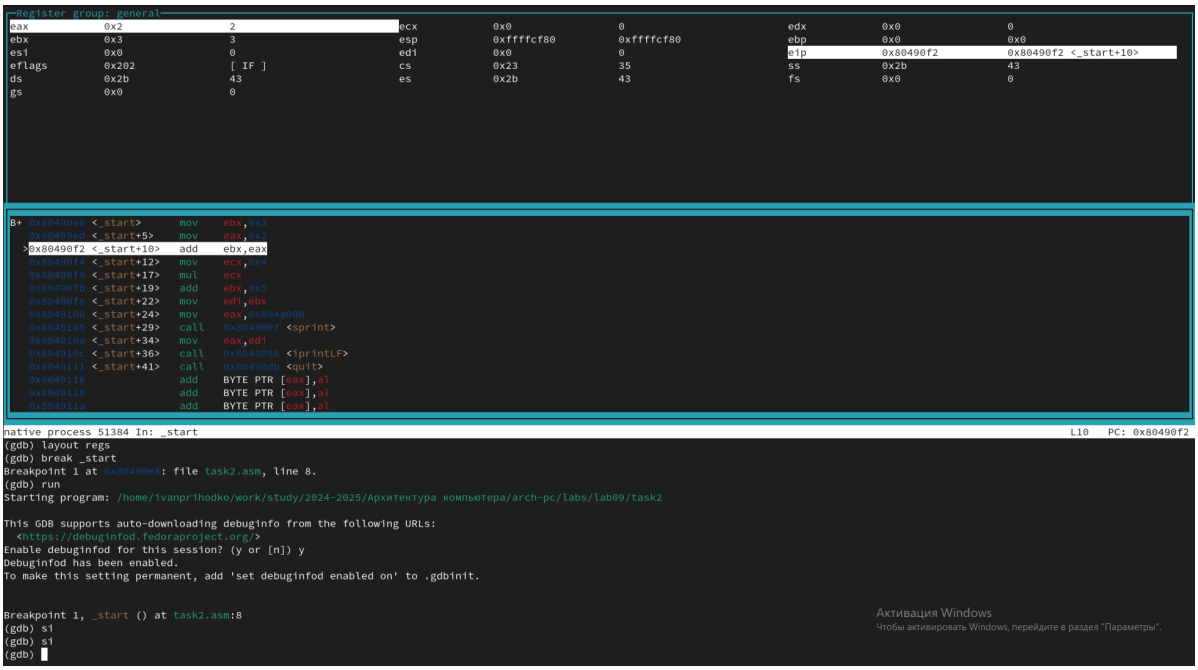

Рис. 3.10: Значение всех регистров на 2 шаге

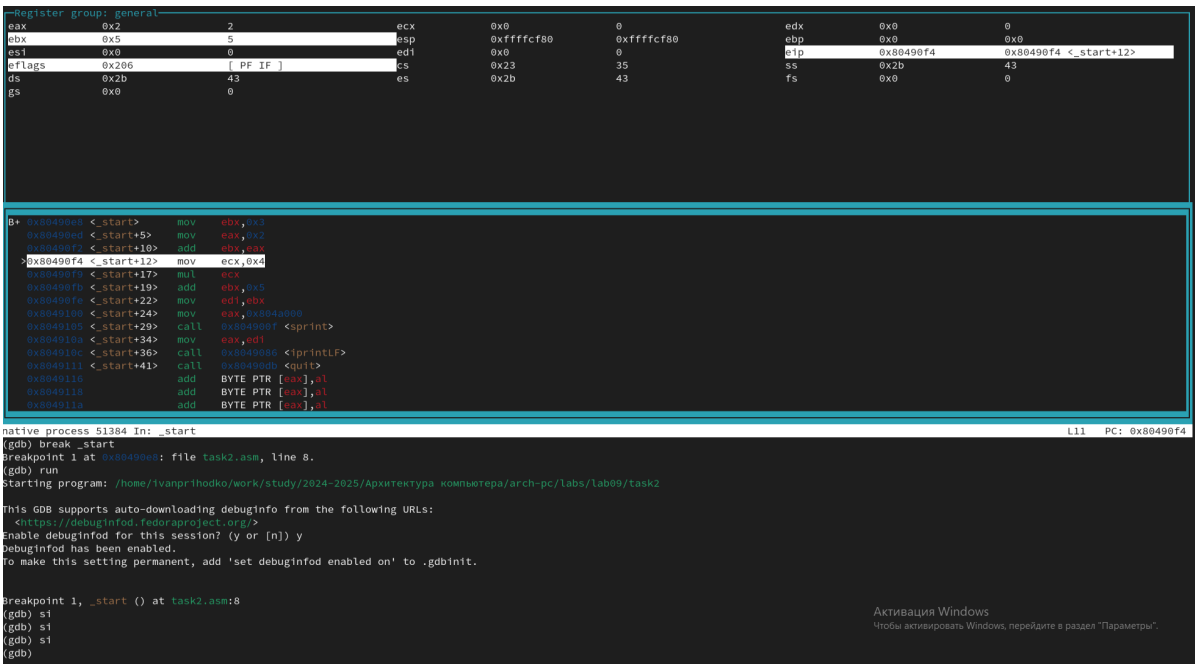

Рис. 3.11: Значение всех регистров на 3 шаге

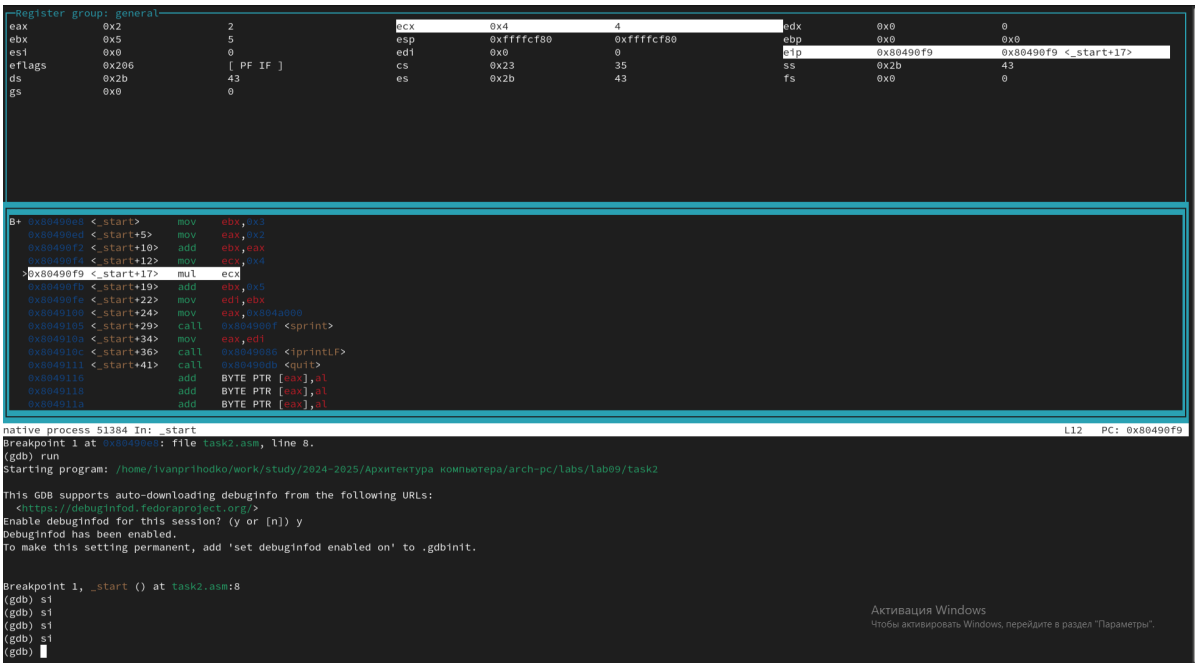

Рис. 3.12: Значение всех регистров на 4 шаге

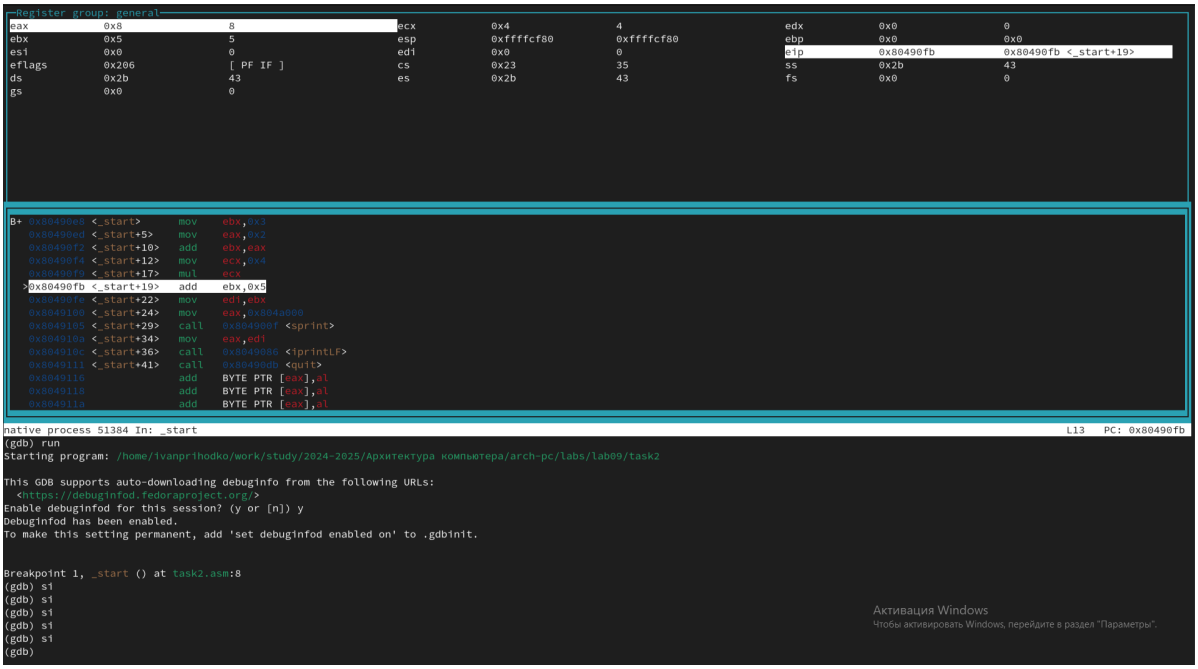

Рис. 3.13: Значение всех регистров на 5 шаге

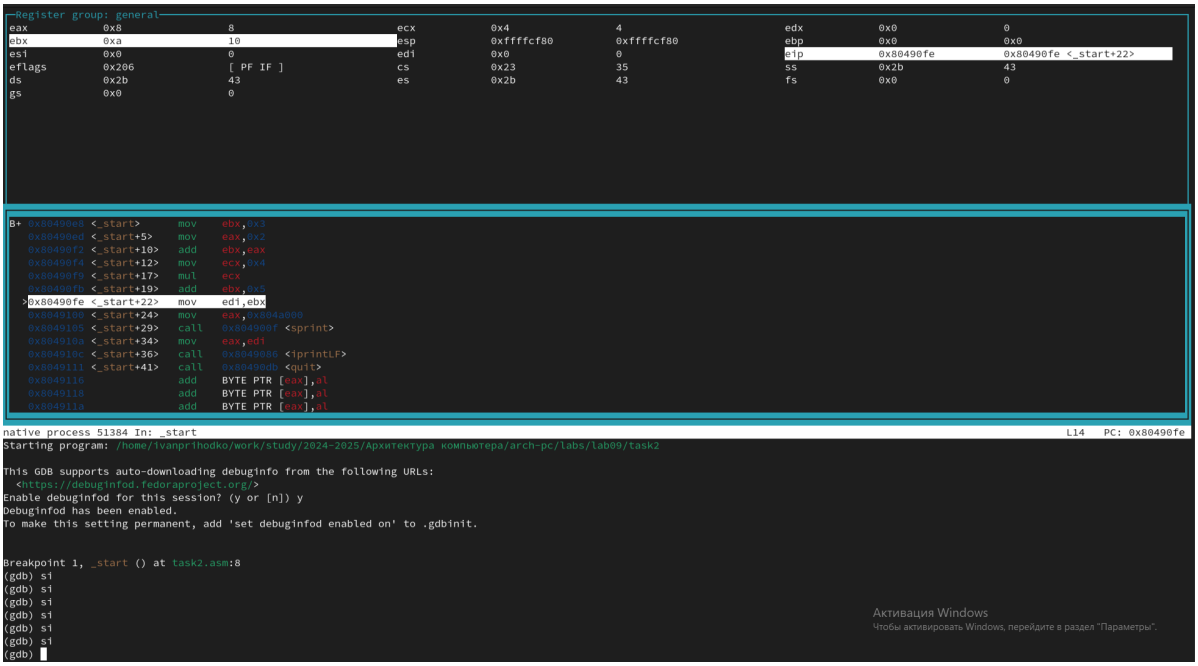

Рис. 3.14: Значение всех регистров на 6 шаге

Как видим, мы должны были умножить значение регистра ebx, но умножили регистр eax. Нам необходимо все результаты хранить в регистре eax. Изменим

код (рис. 3.15).

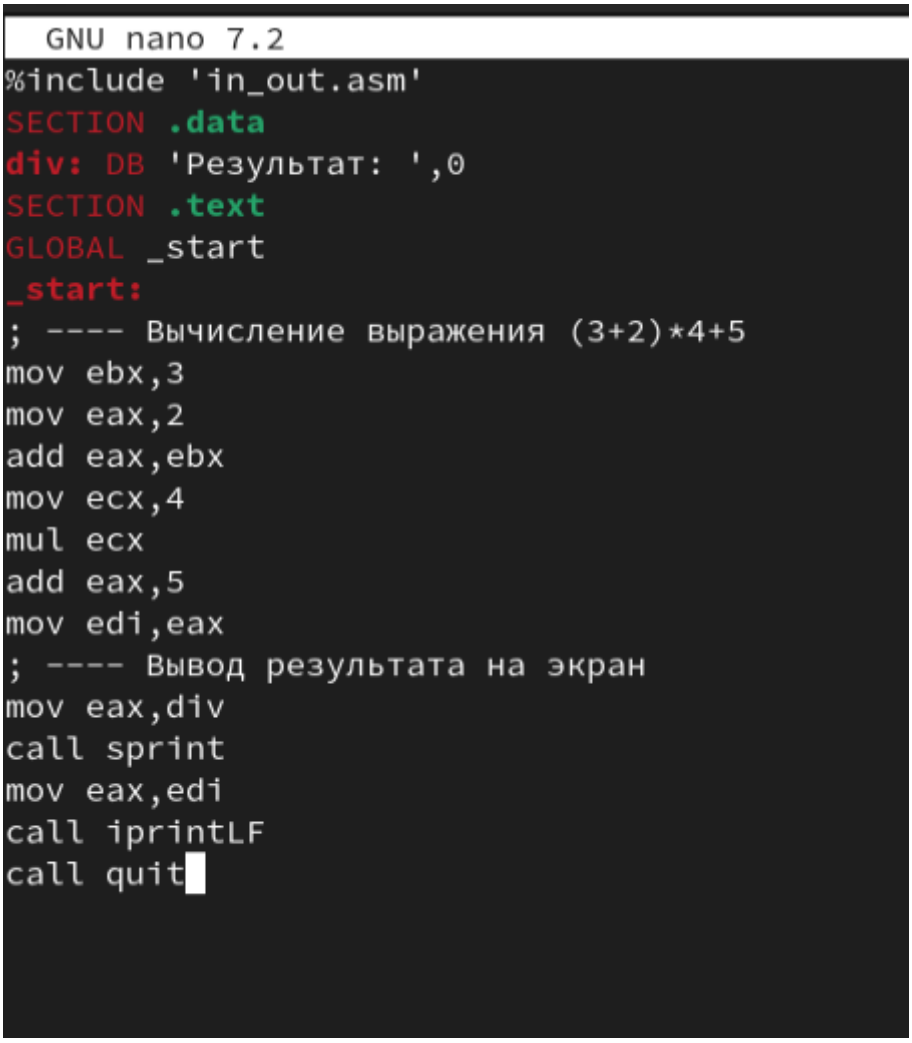

Рис. 3.15: Редактирование кода

Проверим корректность его выполнения (рис. 3.16).

Рис. 3.16: Сборка кода и проверка выполнения

4 Выводы

В результате выполнения лабораторной работы были получены представления о работе подпрограмм, а также было реализовано несколько программ, использующих подпрограммы. Также, были получены навыки работы с базовым функионалом gdb, и с помощью gdb была отловлена ошибка в коде программы.