Отчет по лабораторной работе № 9

Шубина София Антоновна

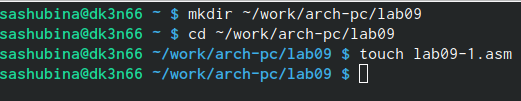
Содержание

# 1 Цель работы

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

# 2 Выполнение лабораторной работы

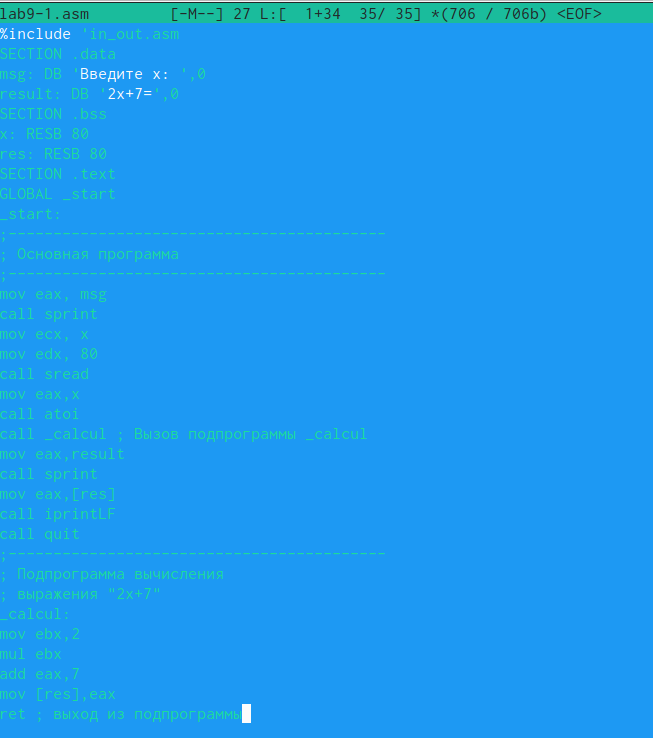
Реализация подпрограмм в NASM 1. Создадим каталог для выполнения лабораторной работы No 9, перейдем в него и со- здадим файл lab09-1.asm: mkdir ~/work/arch-pc/lab09 cd ~/work/arch-pc/lab09 touch lab09-1.asm (рис. ??).



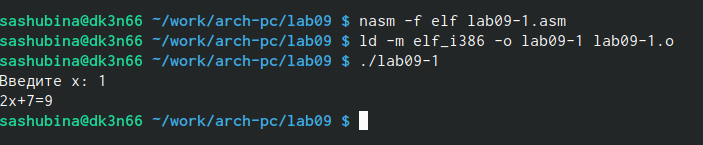
Создание файла и каталога

1. В качестве примера рассмотрим программу вычисления арифметического выражения 𝑓(𝑥) = 2𝑥 + 7 с помощью подпрограммы \_calcul. В данном примере 𝑥 вводится с клавиатуры, а само выражение вычисляется в подпрограмме. Внимательно изучим текст программы. Первые строки программы отвечают за вывод сообщения на экран (call sprint), чтение данных введенных с клавиатуры (call sread) и преобразования введенных данных из символьного вида в численный (call atoi) После следующей инструкции call \_calcul, которая передает управление подпрограмме \_calcul, будут выполнены инструкции подпрограммы Инструкция ret является последней в подпрограмме и ее исполнение приводит к воз- вращению в основную программу к инструкции, следующей за инструкцией call, которая вызвала данную подпрограмму. Последние строки программы реализую вывод сообщения (call sprint), результата вы- числения (call iprintLF) и завершение программы (call quit).

Введем в файл lab09-1.asm текст программы из листинга. Создадим исполняемый файл и проверим его работу. (рис. ??, ??).

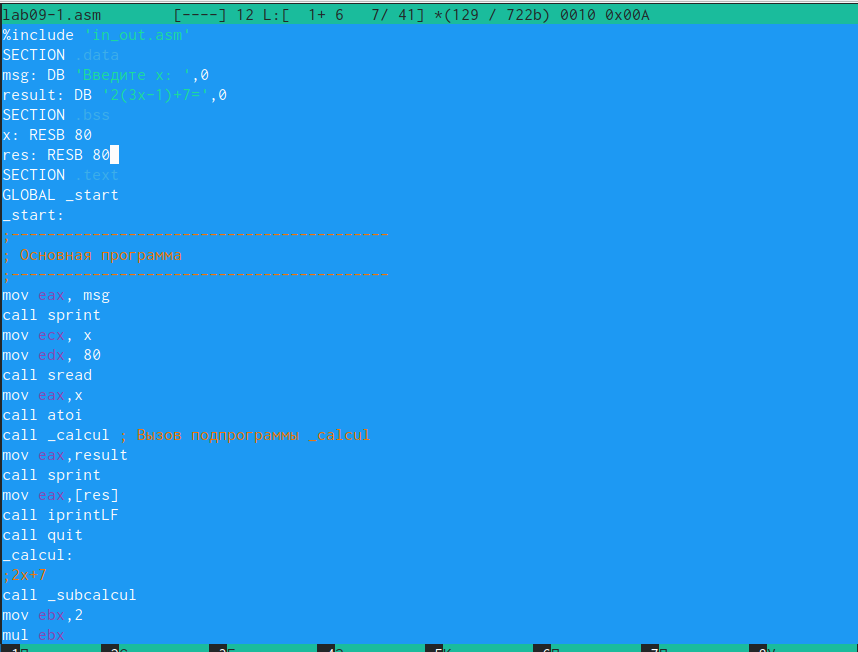
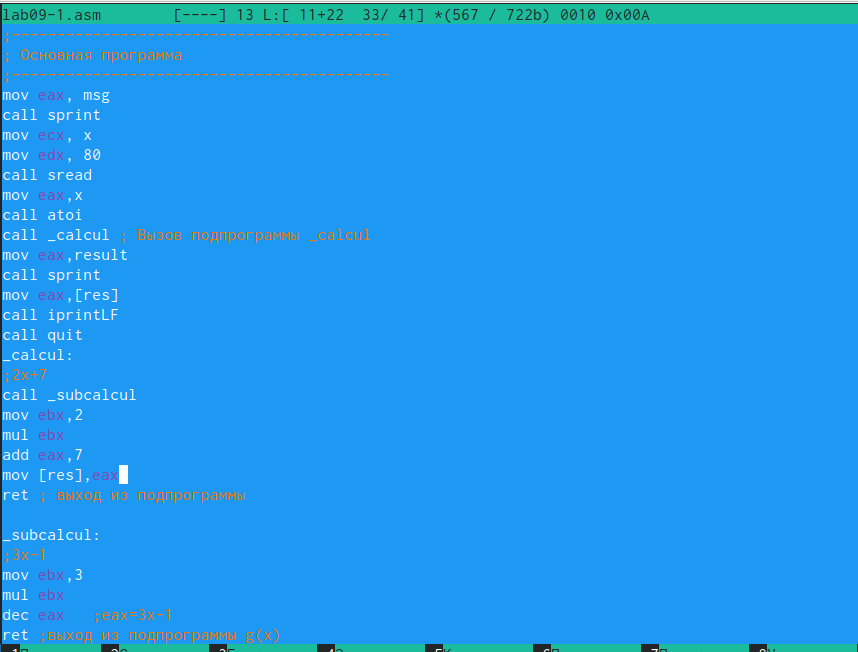
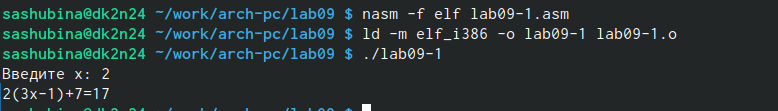


Ввод текста программы



Создание исполняемого файла и проверка его работы

Изменим текст программы, добавив подпрограмму \_subcalcul в подпрограмму \_calcul, для вычисления выражения 𝑓(𝑔(𝑥)), где 𝑥 вводится с клавиатуры, 𝑓(𝑥) = 2𝑥 + 7, 𝑔(𝑥) = 3𝑥 − 1. Т.е. 𝑥 передается в подпрограмму \_calcul из нее в подпрограмму \_subcalcul, где вычисляется выражение 𝑔(𝑥), результат возвращается в \_calcul и вычисляется выражение 𝑓(𝑔(𝑥)). Результат возвращается в основную программу для вывода результата на экран. тладка программам с помощью GDB (рис. ??, ??, ??).

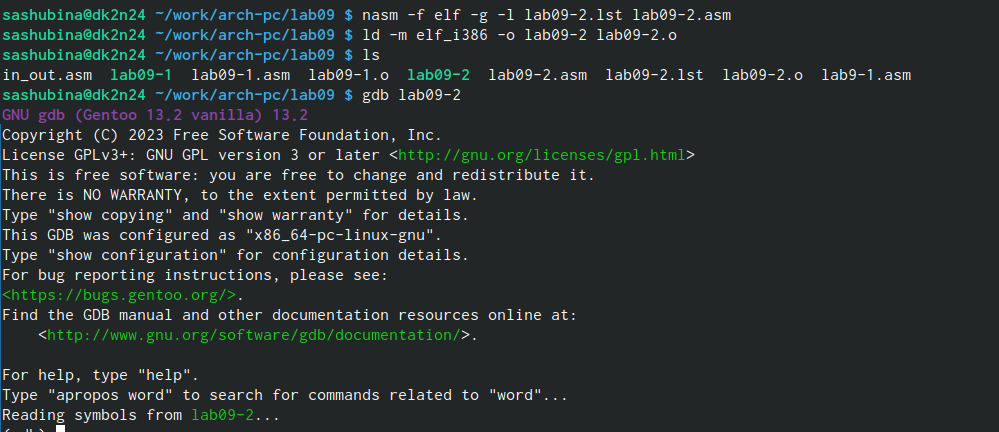
  

Создадим файл lab09-2.asm с текстом программы из Листинга. (Программа печати сообщения Hello world!) (рис. ??).

Создание файла

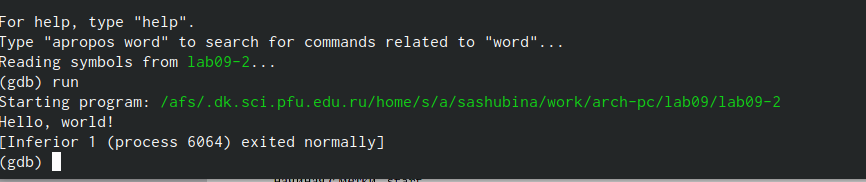
Создание файла

Получим исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию, для этого трансляцию программ необходимо проводить с ключом ‘-g’. nasm -f elf -g -l lab09-2.lst lab09-2.asm ld -m elf\_i386 -o lab09-2 lab09-2.o Загрузим исполняемый файл в отладчик gdb: user@dk4n31:~$ gdb lab09-2 (рис. ??).



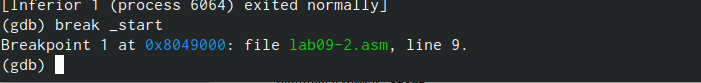
Трансляция программы при помощи ключа -g

Проверим работу программы, запустив ее в оболочке GDB с помощью команды run (со- кращённо r): (gdb) run (рис. ??).



Проверка работы программы

Starting program: ~/work/arch-pc/lab09/lab09-2 Hello, world! [Inferior 1 (process 10220) exited normally] (gdb) Для более подробного анализа программы установите брейкпоинт на метку \_start, с которой начинается выполнение любой ассемблерной программы, и запустите её. (gdb) break \_start (рис. ??).



Запуск исполянемого файла

Breakpoint 1 at 0x8049000: file lab09-2.asm, line 12. (gdb) run Starting program: ~/work/arch-pc/lab09/lab09-2 Breakpoint 1, \_start () at lab09-2.asm:12 12 mov eax, 4

Посмотрим дисассимилированный код программы с помощью команды disassemble начиная с метки \_start (gdb) disassemble \_start (рис. ??).



Просмотр кода программы с помощью команды disassemble

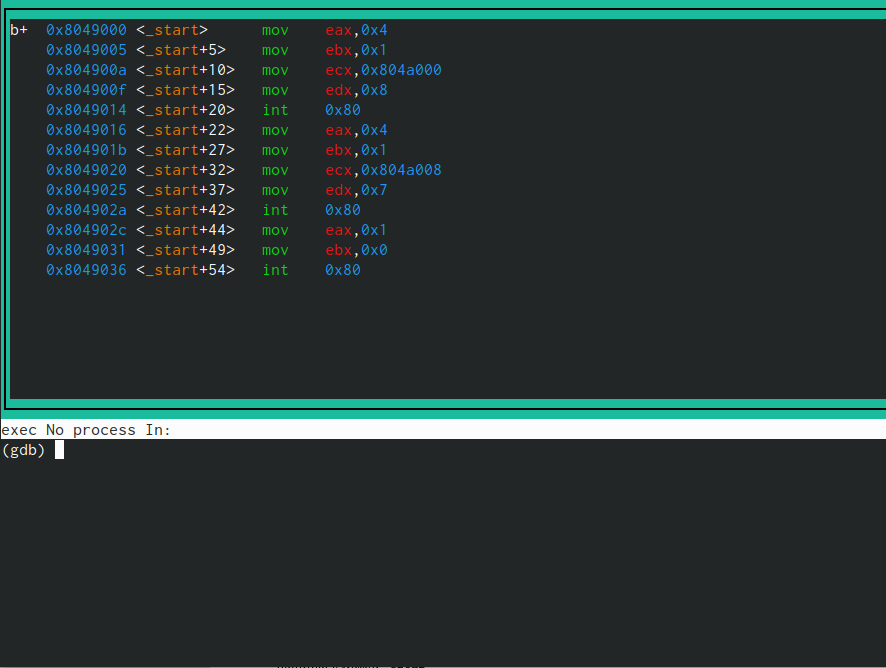
Переключимся на отображение команд с Intel’овским синтаксисом, введя команду set disassembly-flavor intel (gdb) set disassembly-flavor intel (gdb) disassemble \_start (рис. ??).



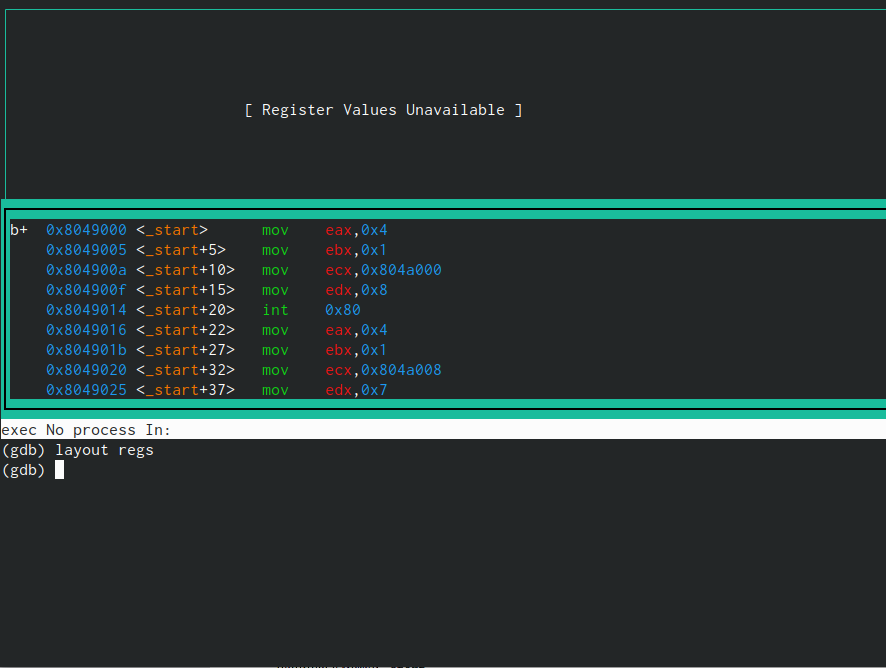
Ввод команды disassembly-flavor intel

Перечислим различия отображения синтаксиса машинных команд в режимах ATT и Intel. - В синтаксисе Intel вводится регистр после команды с помещенным значением, после вводится адрес помещаемого значения или число. В синтаксисе ATT сначала выводится ссылка на заносимое в регистр значение с помощью символа “$”, а после нее регистр с символом.

Включите режим псевдографики для более удобного анализа программы: (gdb) layout asm (gdb) layout regs (рис. ??, ??).



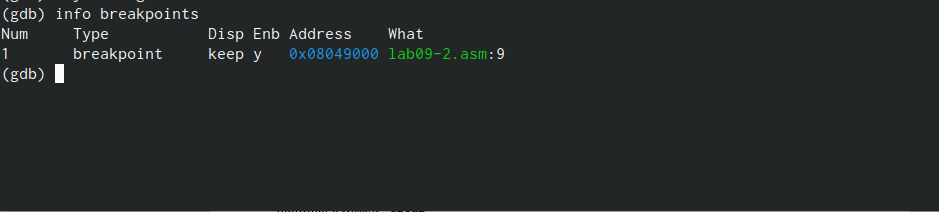
Ввод режима псевдографики(layout asm)



Ввод режима псевдографики(layout regs)

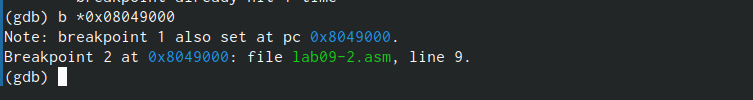
В этом режиме есть три окна: • В верхней части видны названия регистров и их текущие значения; • В средней части виден результат дисассимилирования программы; • Нижняя часть доступна для ввода команд. бавление точек останова Установить точку останова можно командой break (кратко b). Типичный аргумент этой команды — место установки. Его можно задать или как номер строки программы (имеет смысл, если есть исходный файл, а программа компилировалась с информацией об отладке), или как имя метки, или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка»:

На предыдущих шагах была установлена точка останова по имени метки (\_start). Проверим это с помощью команды info breakpoints (кратко i b): (gdb) info breakpoints (рис. ??).

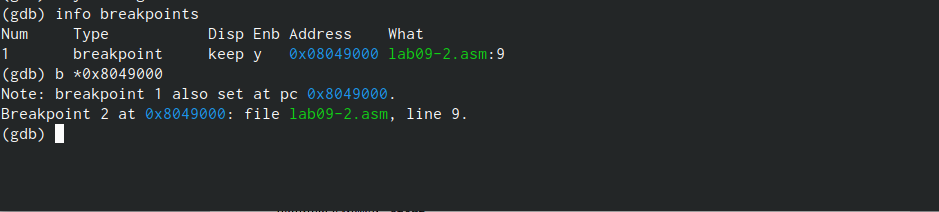


Проверка установки точки остановы по имени метки

Установим еще одну точку останова по адресу инструкции. Адрес инструкции можно увидеть в средней части экрана в левом столбце соответствующей инструкции. Определите адрес предпоследней инструкции (mov ebx,0x0) и установите точку останова. (gdb) break \* Посмотрите информацию о всех установленных точках останова: (gdb) i b (рис. ??, ??).



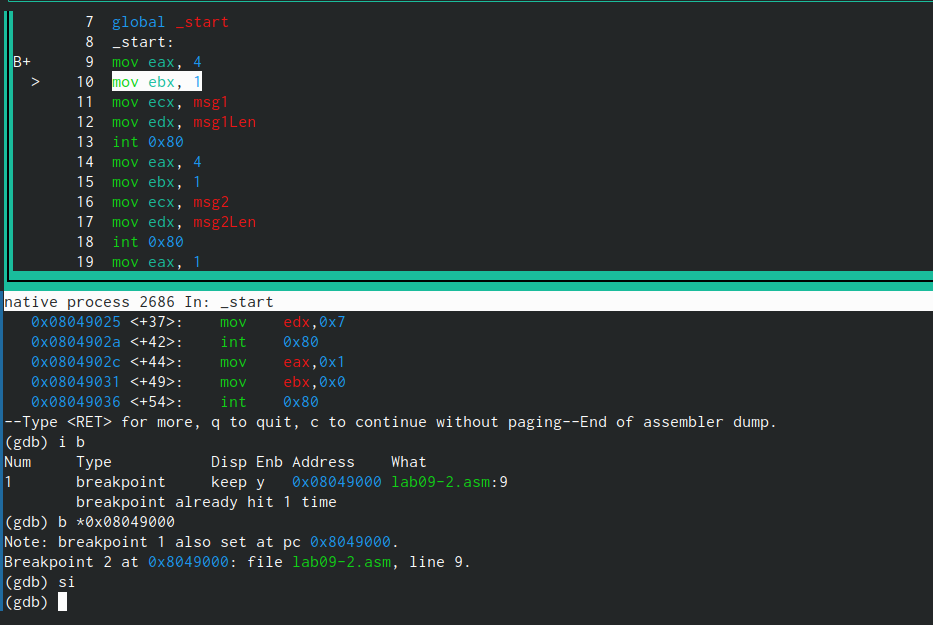
Ввод адреса



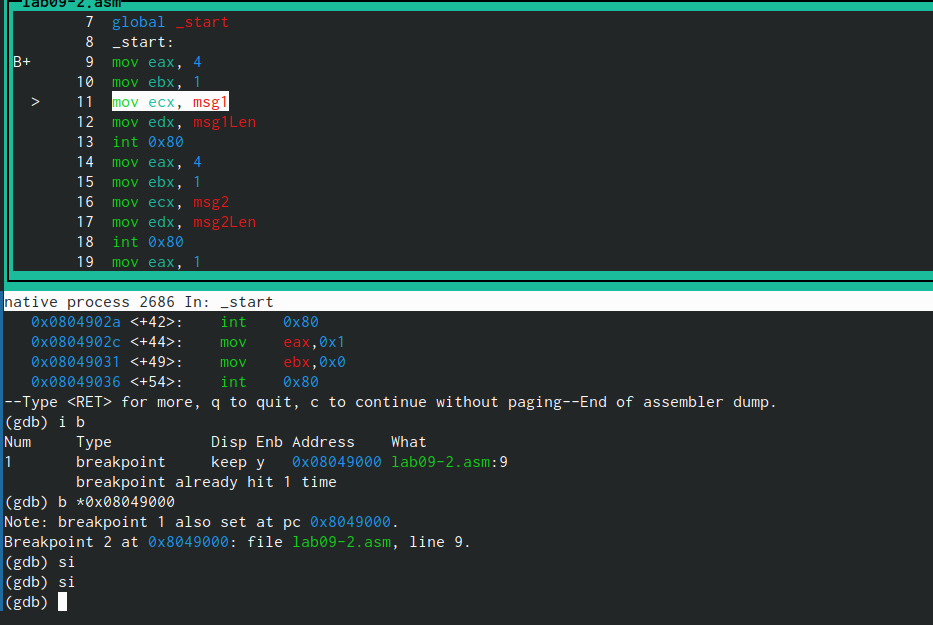
Просмотр информации о всех установленных точках останова

Работа с данными программы в GDB Отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

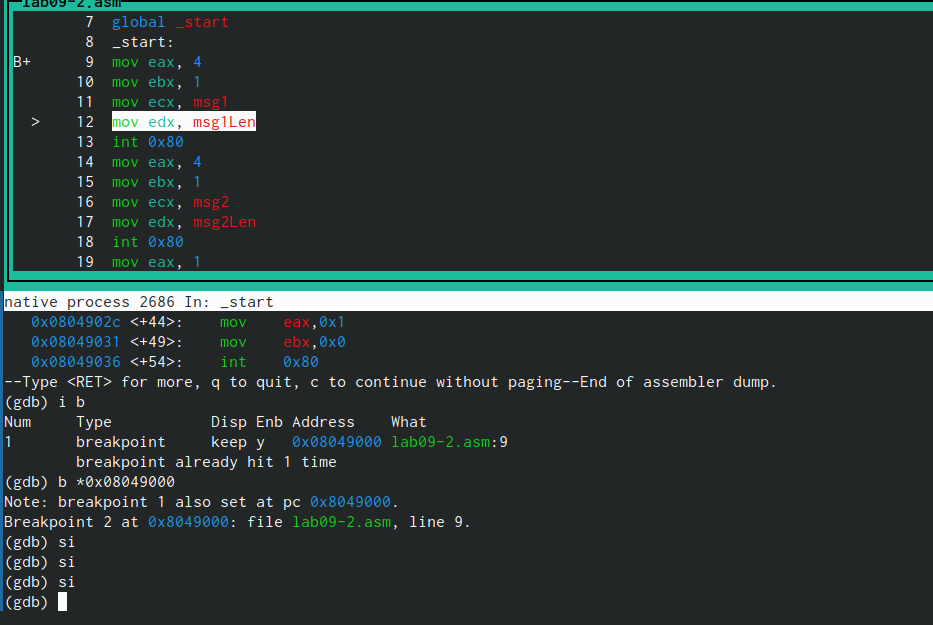
Выполним 5 инструкций с помощью команды stepi (или si) и проследим за изменением значений регистров. Значения каких регистров изменяются? Изменились значения регистров:eax,ecx,edx,ebx, значения остальных регистров не изменились. (рис. ??, ??,??, ??,??).



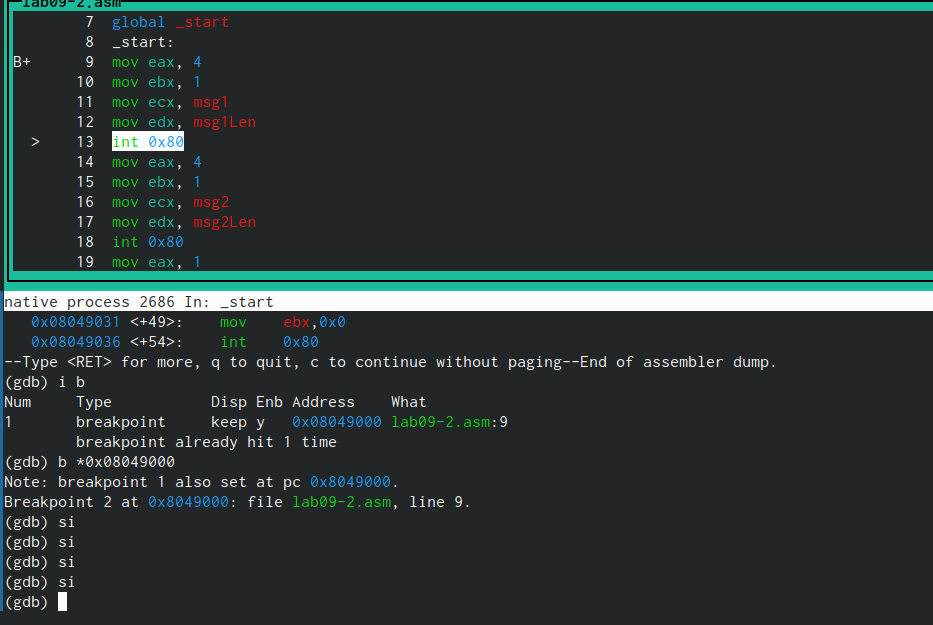
Выполнение команды stepi(si)



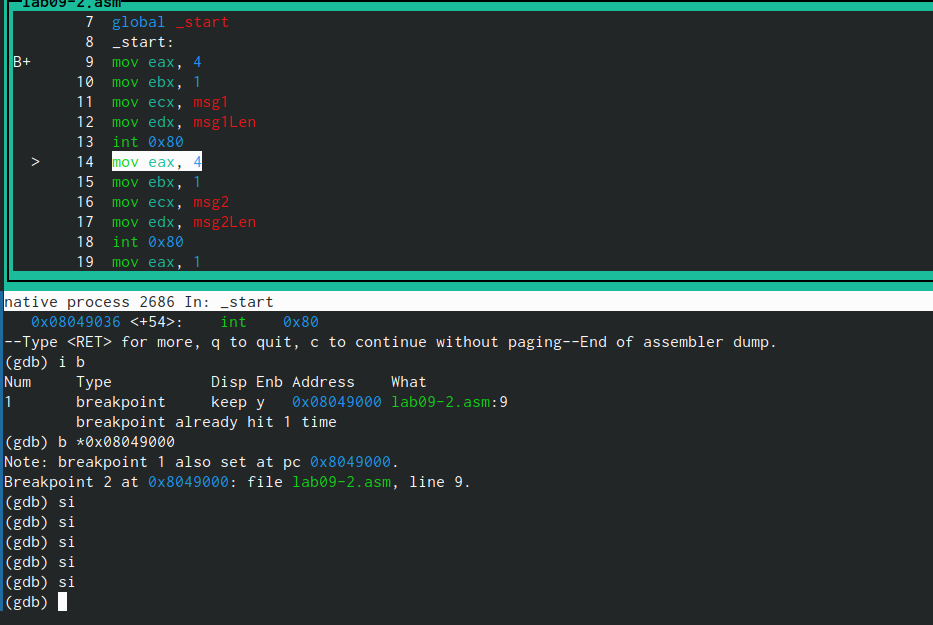
Выполнение команды stepi(si



Выполнение команды stepi(si

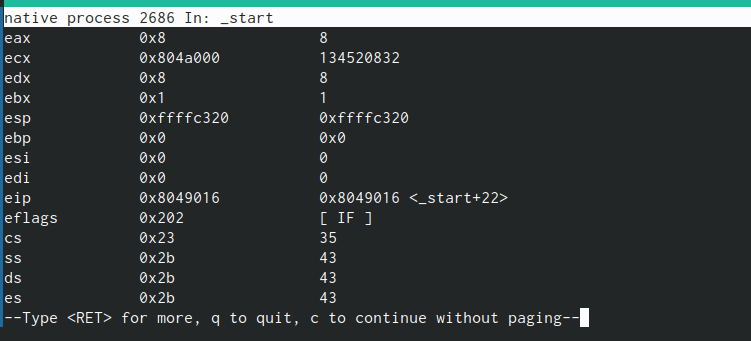


Выполнение команды stepi(si



Выполнение команды stepi(si

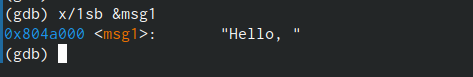
Посмотреть содержимое регистров также можно с помощью команды info registers (или i r). (gdb) info registers (рис. ??).



Просмотр содержимого регистров

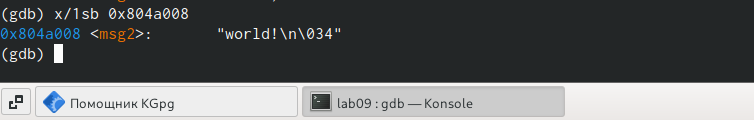
Для отображения содержимого памяти можно использовать команду x , которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: x/NFU . С помощью команды x & также можно посмотреть содержимое пере- менной.

Посмотрим значение переменной msg1 по имени (gdb) x/1sb &msg1 0x804a000 : “Hello,” (рис. ??).



Просмотр значения переменной по имени

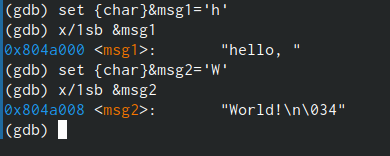
Посмотрите значение переменной msg2 по адресу. Адрес переменной можно определить по дизассемблированной инструкции. Посмотрите инструкцию mov ecx,msg2 которая запи- сывает в регистр ecx адрес перемененной msg2 (рис. ??).



Просмотр значения переменной по адресу

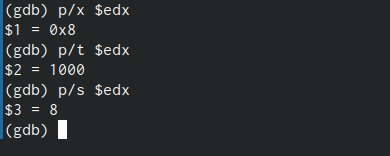
Изменить значение для регистра или ячейки памяти можно с помощью команды set, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс $, а перед адресом нужно указать в фигурных скобках тип данных (раз- мер сохраняемого значения; в качестве типа данных можно использовать типы языка Си).

Изменим первый символ переменной msg1: (gdb) set {char}msg1=‘h’ (gdb) x/1sb &msg1 0x804a000 : “hello,” (gdb) Заменим любой символ во второй переменной msg2. (рис. ??).



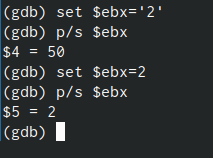
Изменение первого символа в переменной msg1 и замена любого символа в переменной msg2

Чтобы посмотреть значения регистров используется команда print /F (перед име- нем регистра обязательно ставится префикс $): p/F $ (рис. ??).

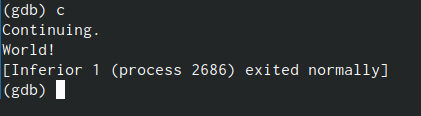


Вывод значения регистра edx

Выведем в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра edx. С помощью команды set изменим значение регистра ebx: (gdb) set $ebx=‘2’ (gdb) p/s $ebx $3 = 50 (gdb) set $ebx=2 (gdb) p/s $ebx $4 = 2 (gdb) (рис. ??).

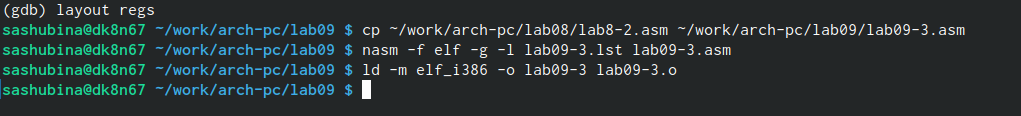
 Объясните разницу вывода команд p/s $ebx. Мы занесли в регистр ebx символ “2”, после значения регистра видим код символа “2”. Если занести в регистр изначально число “2”, а не символ, то команда вывела значение “2”.

Завершим выполнение программы с помощью команды continue (сокращенно c) или stepi (сокращенно si) и выйдем из GDB с помощью команды quit (сокращенно q). (рис. ??).



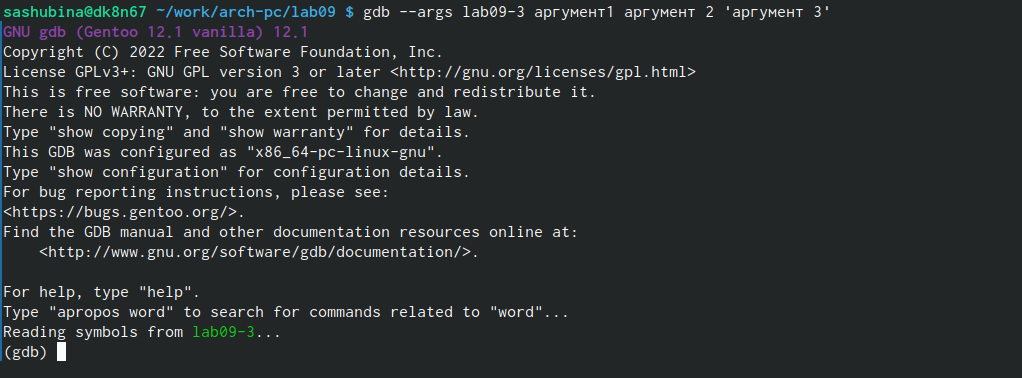
Завершение выполнения работы с помощью команды continue(c)

Обработка аргументов командной строки в GDB Скопируем файл lab8-2.asm, созданный при выполнении лабораторной работы No8, с программой выводящей на экран аргументы командной строки в файл с именем lab09-3.asm: cp ~/work/arch-pc/lab08/lab8-2.asm ~/work/arch-pc/lab09/lab09-3.asm Создайте исполняемый файл. nasm -f elf -g -l lab09-3.lst lab09-3.asm ld -m elf\_i386 -o lab09-3 lab09-3.o (рис. ??).



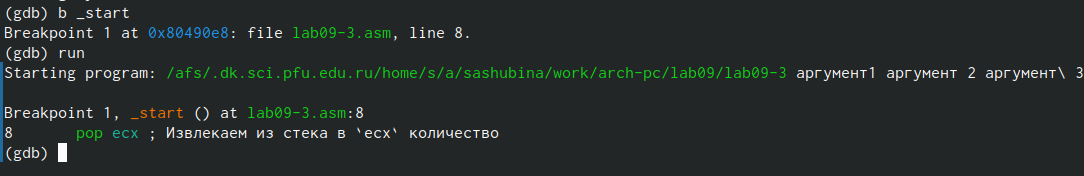
Копирование файла и создане иполняемого файла

Для загрузки в gdb программы с аргументами необходимо использовать ключ –args. Загрузим исполняемый файл в отладчик, указав аргументы: gdb –args lab09-3 аргумент1 аргумент 2 ‘аргумент 3’ (рис. ??).



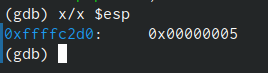
Загрузим исполняемый файл в отладчик,указав аргументы

Как отмечалось в предыдущей лабораторной работе, при запуске программы аргументы командной строки загружаются в стек. Исследуем расположение аргументов командной строки в стеке после запуска программы с помощью gdb. Для начала установим точку останова перед первой инструкцией в программе и запустим ее. (gdb) b \_start (gdb) run (рис. ??).



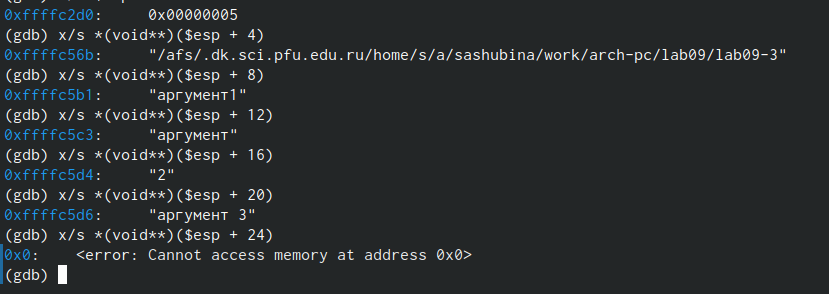
Установка точки и запуск программы

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы): (gdb) x/x $esp 0xffffd200: 0x05 (рис. ??).



Содержимое регистра esp

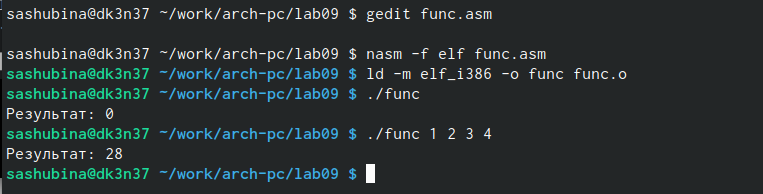
Как видно, число аргументов равно 5 – это имя программы lab09-3 и непосредственно аргументы: аргумент1, аргумент, 2 и ‘аргумент 3’. Посмотриv остальные позиции стека – по адесу [esp+4] располагается адрес в памяти где находиться имя программы, по адесу [esp+8] храниться адрес первого аргумента, по аресу [esp+12] – второго и т.д. (рис. ??).



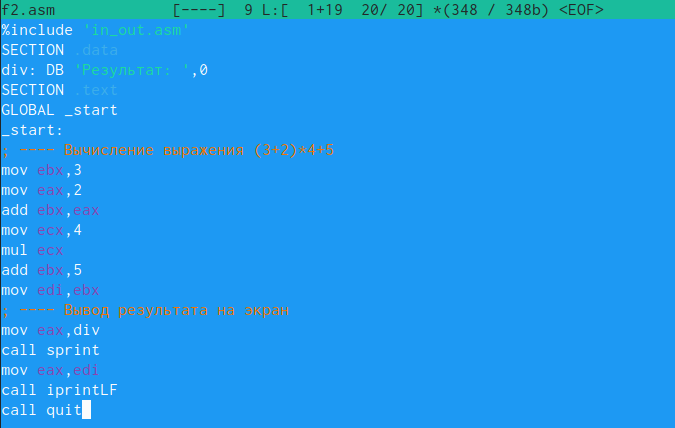
Содержимое регистра esp

Объясните, почему шаг изменения адреса равен 4 ([esp+4], [esp+8], [esp+12] и т.д.). Шаг равен 4 ,потому что при прибавлении значений аргументов в стеке значение регистра esp увеличивается на 4. #Задание для самостоятельной работы 1. Преобразуем программу из лабораторной работы №8 (Задание No1 для самостоятель- ной работы), реализовав вычисление значения функции 𝑓(𝑥) как подпрограмму. (рис. ??).

%include 'in\_out.asm'  
SECTION .data  
msg db "Результат: ",0  
SECTION .text  
global \_start  
  
\_start:  
pop ecx   
pop edx   
sub ecx,1   
mov esi,0  
  
next:  
cmp ecx,0h   
jz \_end   
pop eax   
call atoi  
call \_calcul  
add esi,eax  
loop next  
\_end:  
mov eax, msg  
call sprint  
mov eax, esi   
call iprintLF   
call quit   
  
\_calcul:  
mov ebx,4   
mul ebx  
mov ebx,3  
sub eax,ebx  
ret

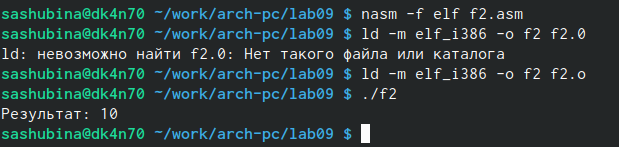
{#fig:044 width=70%

1. В листинге 9.3 приведена программа вычисления выражения (3 + 2) ∗ 4 + 5. При запуске данная программа дает неверный результат. Проверим это. С помощью отладчика GDB, анализируя изменения значений регистров, определим ошибку и исправим ее. Создадим файл f2.asm и впишем в него текст программы вычисления выражения. (рис. ??).



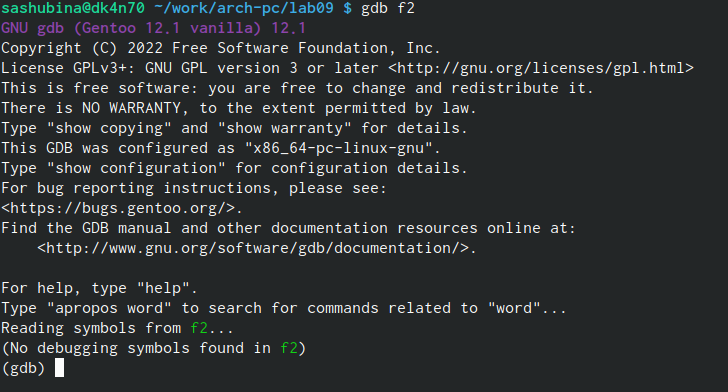
Неверно написанный текст программы

Создадим исполняемый файл и запустим его (рис. ??).



Запуск исполняемого файла

При запуске програма дает неверный результат. С помощью отладчика GDB проанализируем изменения значений регистров, исправим ошибку. Загрузим исполняемый файл в отладчик. (рис. ??).



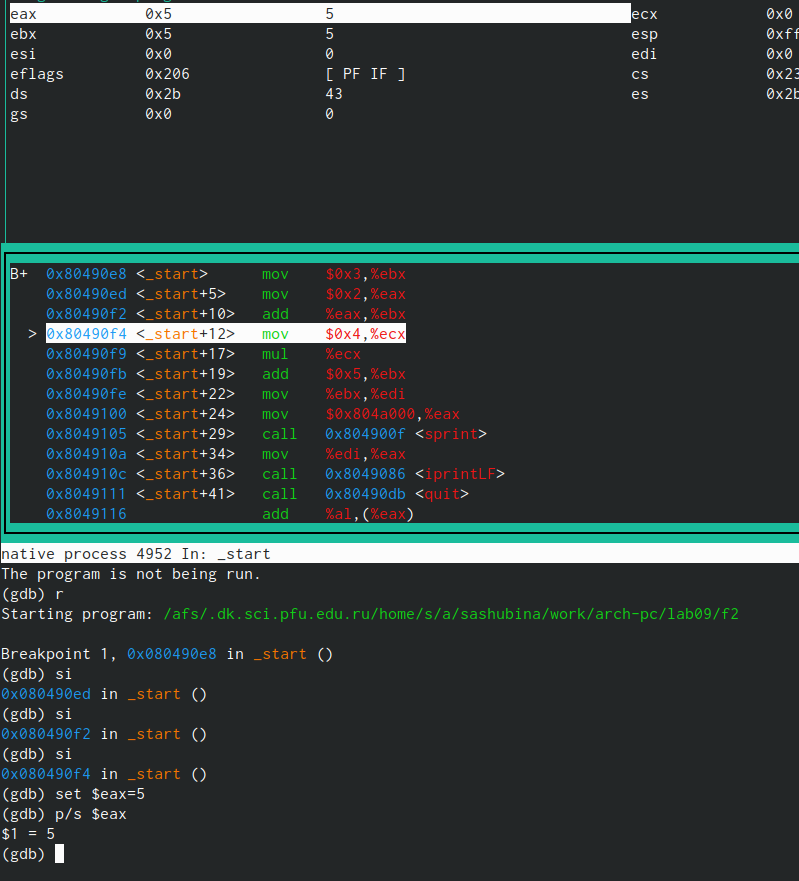
Загрузка файла f2.asm

Просмотр дисассемилированного кода программы (рис. ??).



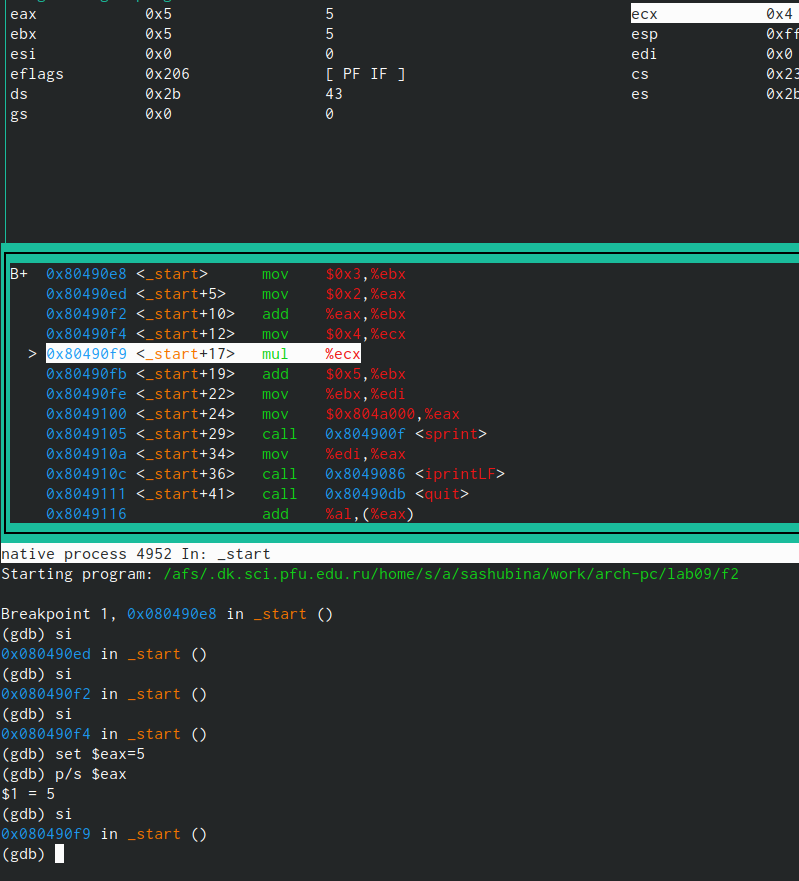
Загрузка файла f2.asm

Включим режим псевдографики, с помощью команды break установим точку основа на иструкции add,ebx,eax. С помощью команды si перейдем к следующей инструкции и проследим за изменением значений регистров. (рис. ??).



Просмотр значений регистров

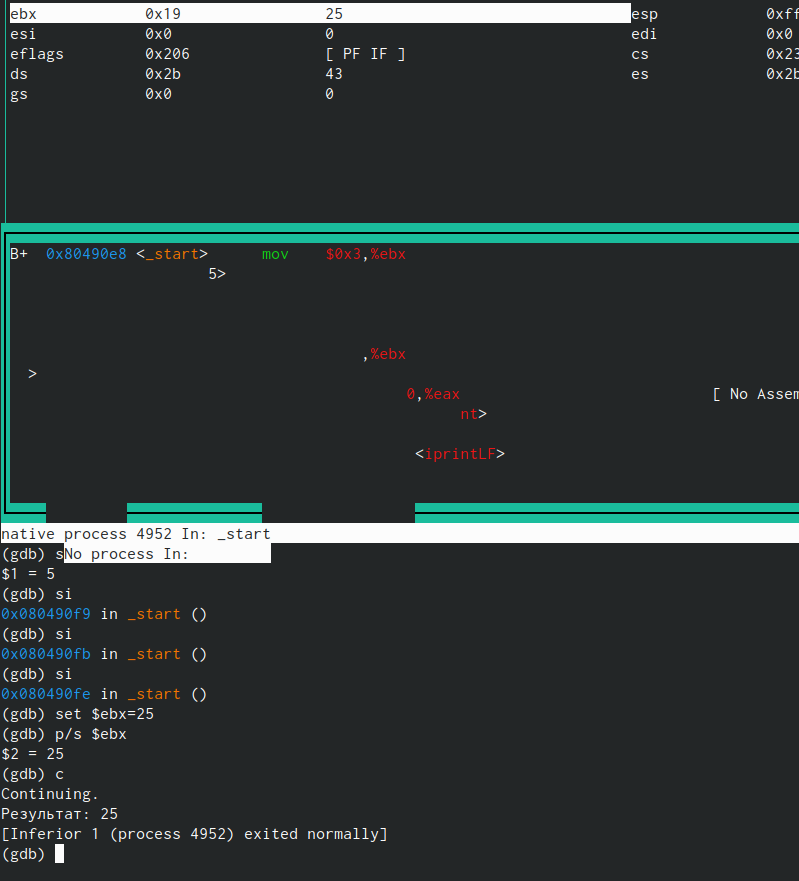
Чтобы исправить программу, занесем в eax с помощью команды set значение 5. (рис. ??).



Изменение значения регистра

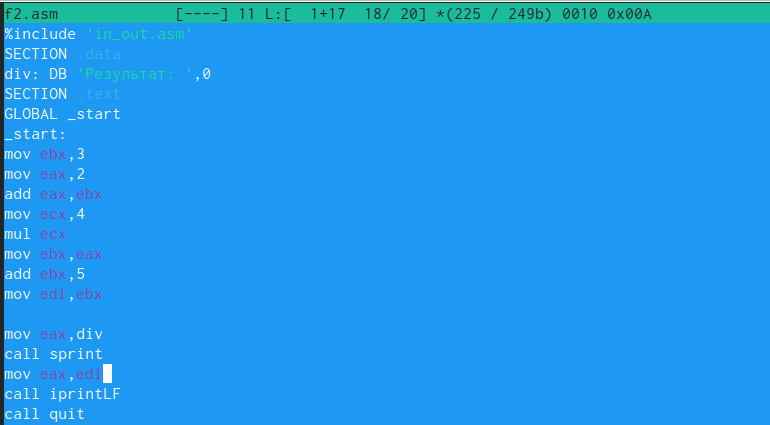
За результат программы отвечает регистр ebx. В него посместим значение 20+5 и запустим программу.

(рис. ??).



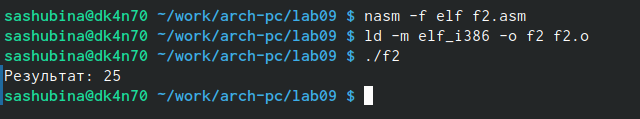
Изменение регистра программы и завершение программы

Изменим текст программы. (рис. ??).



Изменение текста программы

Запустим исполняемый файл, программа выдает верный результат. (рис. ??).



Запуск исполняемого файла

# 3 Выводы

Я приобрела навыки написания программ с использованием подпрограмм. Ознакомилась с методами отладки при помощи GDB и его основными возможностями.

# Список литературы

1. GDB: The GNU Project Debugger. — URL: https://www.gnu.org/software/gdb/.
2. GNU Bash Manual. — 2016. — URL: https://www.gnu.org/software/bash/manual/.
3. Midnight Commander Development Center. — 2021. — URL: https://midnight-commander. org/.
4. NASM Assembly Language Tutorials. — 2021. — URL: https://asmtutor.com/.
5. Newham C. Learning the bash Shell: Unix Shell Programming. — O’Reilly Media, 2005. — 354 с. — (In a Nutshell). — ISBN 0596009658. — URL: http://www.amazon.com/Learning- bash-Shell-Programming-Nutshell/dp/0596009658.
6. Robbins A. Bash Pocket Reference. — O’Reilly Media, 2016. — 156 с. — ISBN 978-1491941591.
7. The NASM documentation. — 2021. — URL: https://www.nasm.us/docs.php.
8. Zarrelli G. Mastering Bash. — Packt Publishing, 2017. — 502 с. — ISBN 9781784396879.
9. Колдаев В. Д., Лупин С. А. Архитектура ЭВМ. — М. : Форум, 2018.
10. Куляс О. Л., Никитин К. А. Курс программирования на ASSEMBLER. — М. : Солон-Пресс,
11. Новожилов О. П. Архитектура ЭВМ и систем. — М. : Юрайт, 2016.
12. Расширенный ассемблер: NASM. — 2021. — URL: https://www.opennet.ru/docs/RUS/nasm/.
13. Робачевский А., Немнюгин С., Стесик О. Операционная система UNIX. — 2-е изд. — БХВ- Петербург, 2010. — 656 с. — ISBN 978-5-94157-538-1.
14. Столяров А. Программирование на языке ассемблера NASM для ОС Unix. — 2-е изд. — М. : МАКС Пресс, 2011. — URL: http://www.stolyarov.info/books/asm\_unix.
15. Таненбаум Э. Архитектура компьютера. — 6-е изд. — СПб. : Питер, 2013. — 874 с. — (Классика Computer Science).
16. Таненбаум Э., Бос Х. Современные операционные системы. — 4-е изд. — СПб. : Питер,
17. — 1120 с. — (Классика Computer Science).