Отчёт по лабораторной работе №9

Дисциплина: Архитектура компьютера

Студент: Верниковская Екатерина Андреевна

Содержание

# Цель работы

Приобрести навыки написания программ с использованием подпрограмм. Познакомиться с методами отладки при помощи GDB и его основными возможностями.

# Задание

1. Создать каталог для программ лабораторной работы №9 и в нём создать файл «lab9-1.asm».
2. Ввести в файл «lab9-1.asm» определённый текст программы с подпрограммой \_calcul. Создать исполняемый файл и запустить его.
3. Изменить текст программы, добавив подпрограмму \_subcalcul. Снова создать исполняемый файл и запустить его.
4. Опять изменить текст программы, добавив команды push и pop, создать исполняемый файл и запустить его.
5. Создать файл «lab9-2.asm» и ввести в него определённый текст программы (Программа печати сообщения Hello world!).
6. Создать исполняемый файл с отладочной информацией.
7. Сделать задания, которые помогут лучше понять структуру GDB.
8. Преобразовать программу из лабораторной работы №8 (задание №1 для самостоятельной работы), реализовав вычисление значения функции f(x) как подпрограмму.

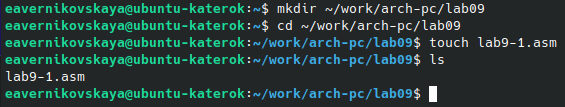
Задание №1 для самостоятельной работы: Написать программу, которая находит сумму значений функции f(x) для x = x1, x2, …, xn. Программа должна выводить значение f(x1) + f(x2) + … + f(xn). Вид функции брать из определённой таблицы, в соответствии с полученным вариантом (В нашем случае это 17 вариант).

1. Создать файл и ввести в него программу, которая вычисляет выражение (3+2)\*4+5. Спомощью отладчика GDB определить ошибку и исправить её.

# Выполнение лабораторной работы

## Реализация подпрограмм в NASM

В созданном каталоге «~/work/arch-pc/lab09» создаём файл «lab9-1.asm» (рис. [-@fig:001])



Создание первого файла

Копируем из каталога «~/work/arch-pc/lab08» файл «in\_out.asm» (рис. [-@fig:002])

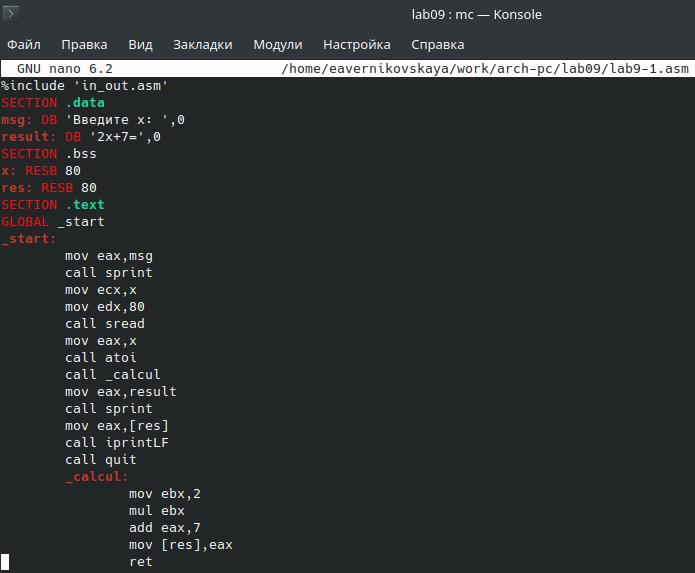


Копирование файла «in\_out.asm»

Вводим нужный текст программы, которая вычисляет арифмитическое выражение f(x)=2x+7 с помощью подпрограммы \_calcul (рис. [-@fig:003])

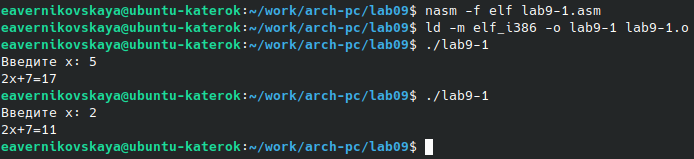
Текст программы:

%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:  
 mov eax,msg  
 call sprint  
 mov ecx,x  
 mov edx,80  
 call sread  
 mov eax,x  
 call atoi  
 call \_calcul  
 mov eax,result  
 call sprint  
 mov eax,[res]  
 call iprintLF  
 call quit  
 \_calcul:  
 mov ebx,2  
 mul ebx  
 add eax,7  
 mov [res],eax  
 ret



Ввод текста программы с подпрограммой \_calcul

Создаём исполняемый файл и запускаем его (рис. [-@fig:004])

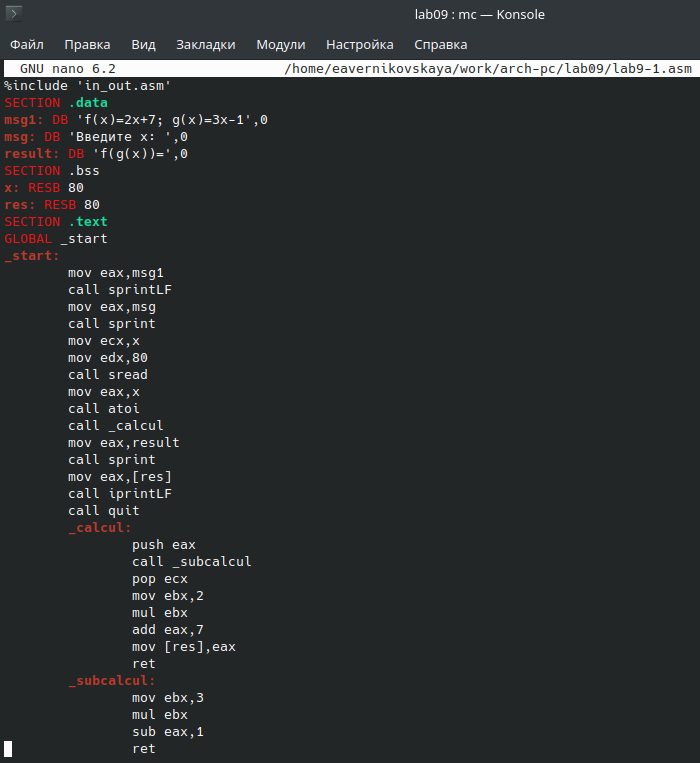


Создание исполняемого файла и его запуск

Изменяем текст программы, добавив подпрограмму \_subcalcul в подпрограмму \_calcul, для вычисления выражения f(g(x)). f(x)=2x+7 а g(x)=3x-1 (рис. [-@fig:005])

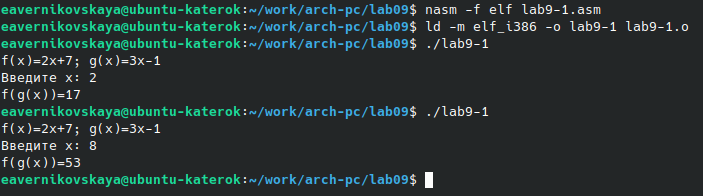
Изменённый текст программы:

%include 'in\_out.asm'  
SECTION .data  
msg1: DB 'f(x)=2x+7; g(x)=3x-1',0  
msg: DB 'Введите x: ',0  
result: DB 'f(g(x))=',0  
SECTION .bss  
x: RESB 80  
res: RESB 80  
SECTION .text  
GLOBAL \_start  
\_start:  
 mov eax,msg1  
 call sprintLF  
 mov eax,msg  
 call sprint  
 mov ecx,x  
 mov edx,80  
 call sread  
 mov eax,x  
 call atoi  
 call \_calcul  
 mov eax,result  
 call sprint  
 mov eax,[res]  
 call iprintLF  
 call quit  
 \_calcul:  
 push eax  
 call \_subcalcul  
 pop ecx  
 mov ebx,2  
 mul ebx  
 add eax,7  
 mov [res],eax  
 ret  
 \_subcalcul:  
 mov ebx,3  
 mul ebx  
 sub eax,1  
 ret



Изменение программы

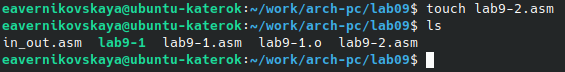
Снова создаём исполняемый файл и запускаем его (рис. [-@fig:006])



Исполняемый файл + запуск

## Отладка программ с помощью GDB

Создаём файл «lab9-2.asm» (рис. [-@fig:007])

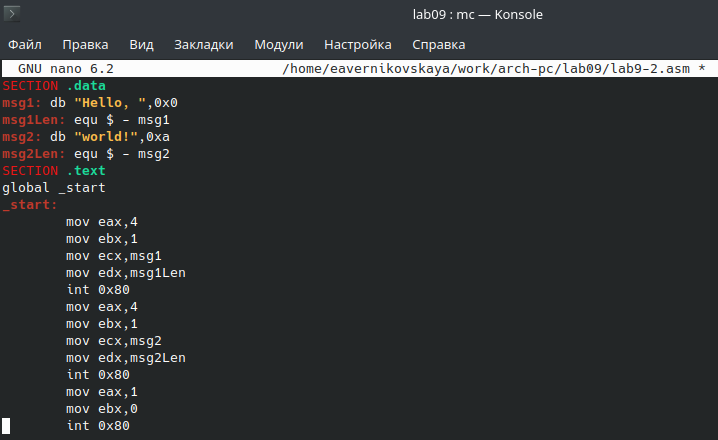


Создание файла «lab9-2.asm»

Вводим текст программы - программа печати сообщения Hello world! (рис. [-@fig:008])

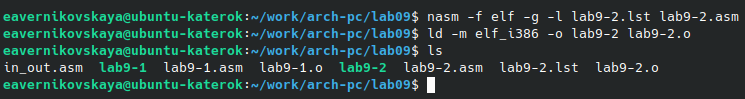
Текст программы:

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



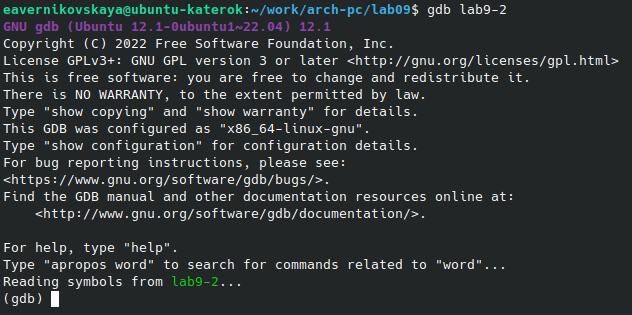
Ввод текста программы

Создаём исполняемый файл. Для работы с GDB в исполняемый файл необходимо добавить отладочную информацию. Для этого трансляцию программ мы проводим с ключом ‘-g’ (рис. [-@fig:009])



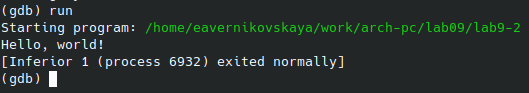
Создание исполняемого файла с ключом ‘-g’

Загружаем исполняемый файл в отладчик gdb (рис. [-@fig:010])



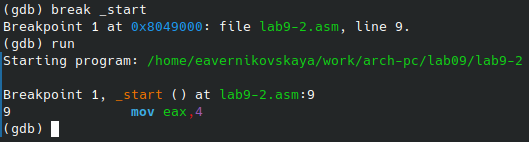
Загрузка исполняемого файла в gdb

Проверяем работу программы, запустив её в оболочке GDB с помощью команды ‘run’ (рис. [-@fig:011])



Проверка работы программы в оболочке gdb

Для более подробного анализа программы устанавливаем брейкпоинт на метку «\_start», с которой начинается выполнение любой ассемблерной программы, и запускаем её (рис. [-@fig:012])



Установка брейкпоинта и запуск

Смотрим дисассимилированный код программы с помощью команды ‘disassemble’ начиная с метки «\_start» (рис. [-@fig:013])



Дисассимилированный код

Переключаемся на отображение команд с Intel’овским синтаксисом, введя команду ’set disassembly-flavor intel’ (рис. [-@fig:014])

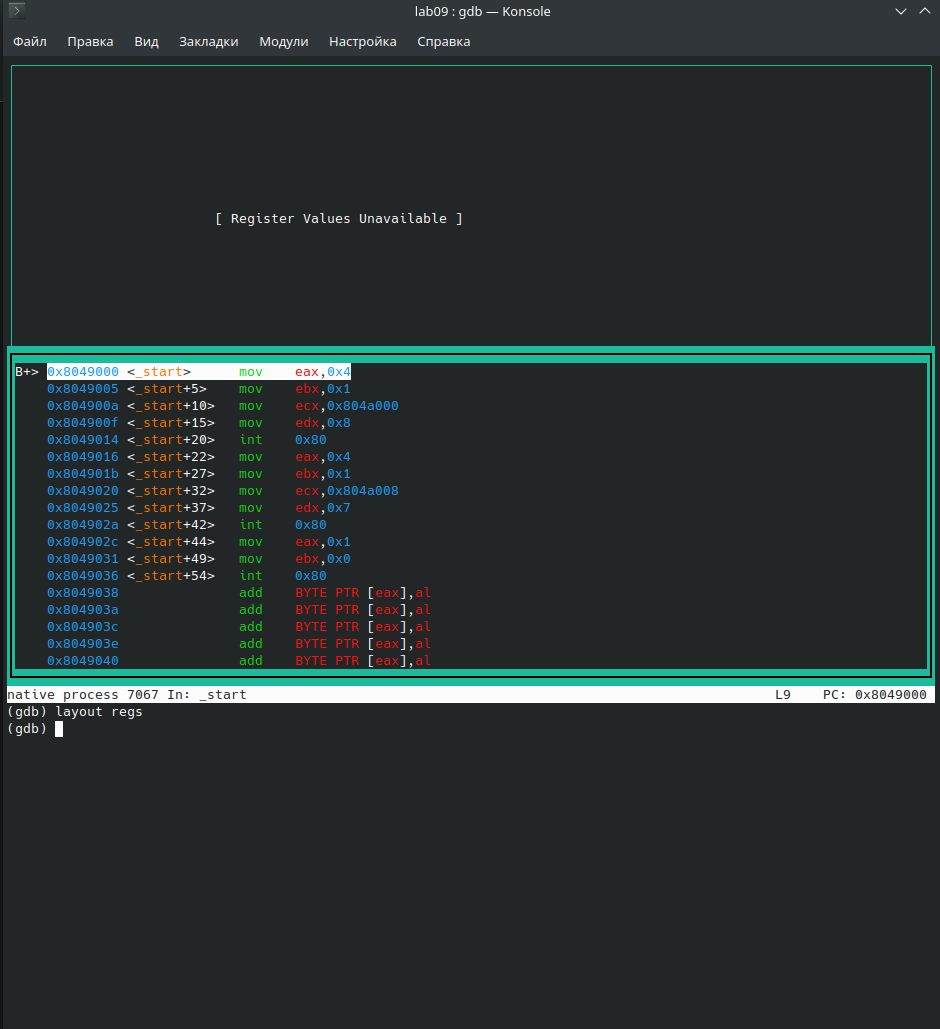


Отображение команд с Intel’овским синтаксисом

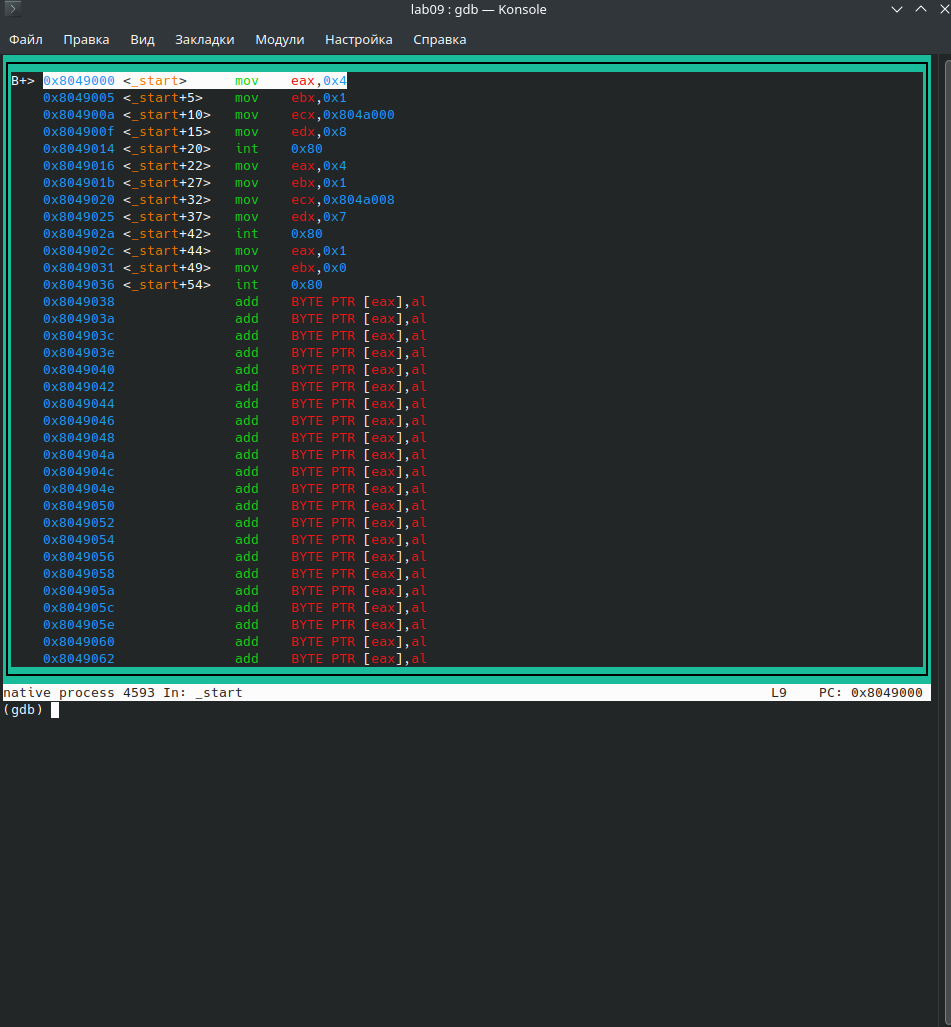
Различия отображения синтаксиса машинных команд в режимах ATT и Intel:

1. **Порядок операндов:**
   * В ATT синтаксисе порядок операндов обратный, сначала указывается исходный операнд, а затем - результирующий операнд.
   * В Intel синтаксисе порядок обычно прямой, результирующий операнд указывается первым, а исходный - вторым.
2. **Разделители:**
   * В ATT синтаксисе разделители операндов - запятые.
   * В Intel синтаксисе разделители могут быть запятые или косые черты (/).
3. **Префиксы размера операндов:**
   * В ATT синтаксисе размер операнда указывается перед операндом с использованием префиксов, таких как “b” (byte), “w” (word), “l” (long) и “q” (quadword).
   * В Intel синтаксисе размер операнда указывается после операнда с использованием суффиксов, таких как “b”, “w”, “d” и “q”.
4. **Знак операндов:**
   * В ATT синтаксисе операнды с позитивными значениями предваряются символом “$”.
   * В Intel синтаксисе операнды с позитивными значениями могут быть указаны без символа “$”.
5. **Обозначение адресов:**
   * В ATT синтаксисе адреса указываются в круглых скобках.
   * В Intel синтаксисе адреса указываются без скобок.
6. **Обозначение регистров:**
   * В ATT синтаксисе обозначение регистра начинается с символа “%”.
   * В Intel синтаксисе обозначение регистра может начинаться с символа “R” или “E” (например, “%eax” или “RAX”).

Включаем режим псевдографики для более удобного анализа программы (рис. [-@fig:015]), (рис. [-@fig:016]



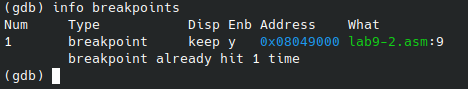
Использование команды ‘layout asm’



Использование команды ‘layout regs’

### Добавление точек останова

Проверяем с помощью команды ‘info breakpoints’ была ли установлена точка останова по имени на предыдущих шагах «\_start» (рис. [-@fig:017])



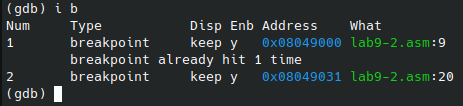
Использование команды ‘info breakpoints’

Определяем адрес предпоследней инструкции (mov ebx,0x0) и устнавливаеи точку останова (рис. [-@fig:018])

Установка точки останова

Установка точки останова

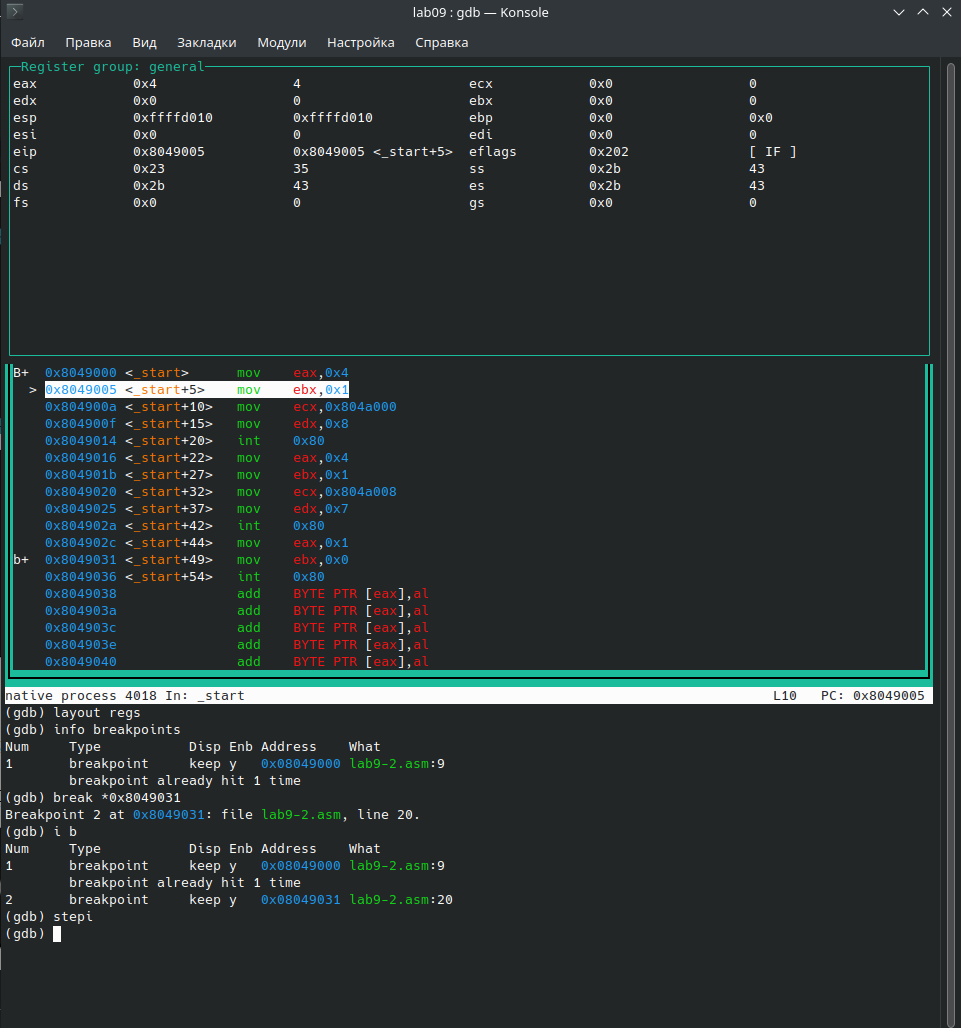
Смотрим информацию о всех установленных точках останова (рис. [-@fig:019])



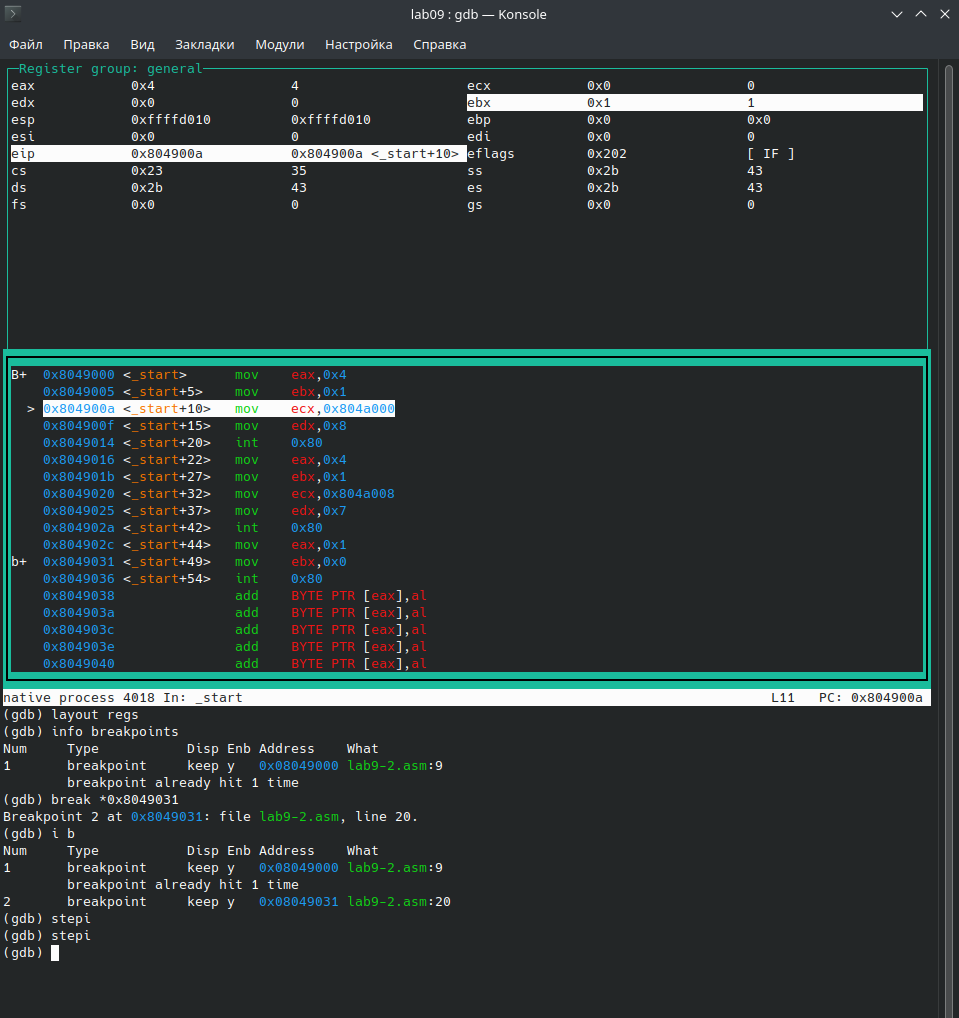
Информация о всех точек останова

### Работа с данными программы в GDB

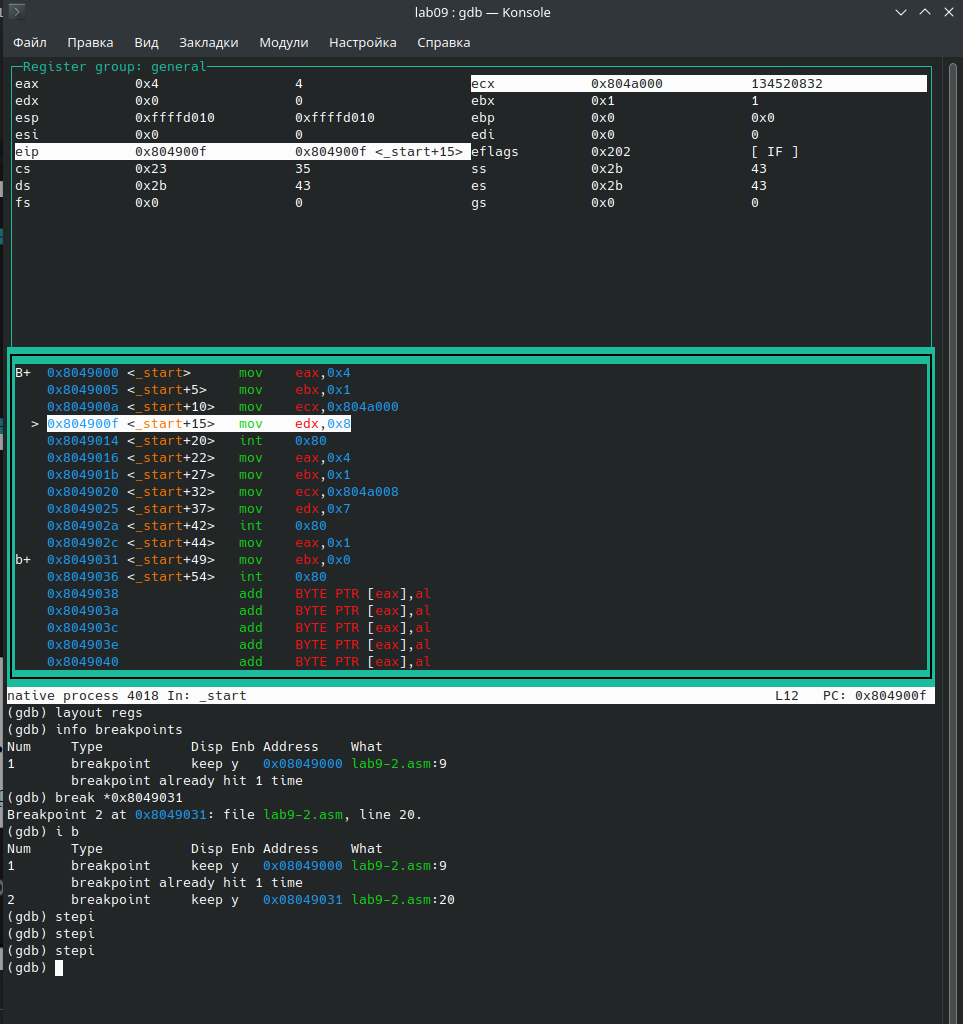
Выполняем 5 инструкций с помощью команды ‘stepi’ (рис. [-@fig:020]), (рис. [-@fig:021]), (рис. [-@fig:022]), (рис. [-@fig:023]), (рис. [-@fig:024])



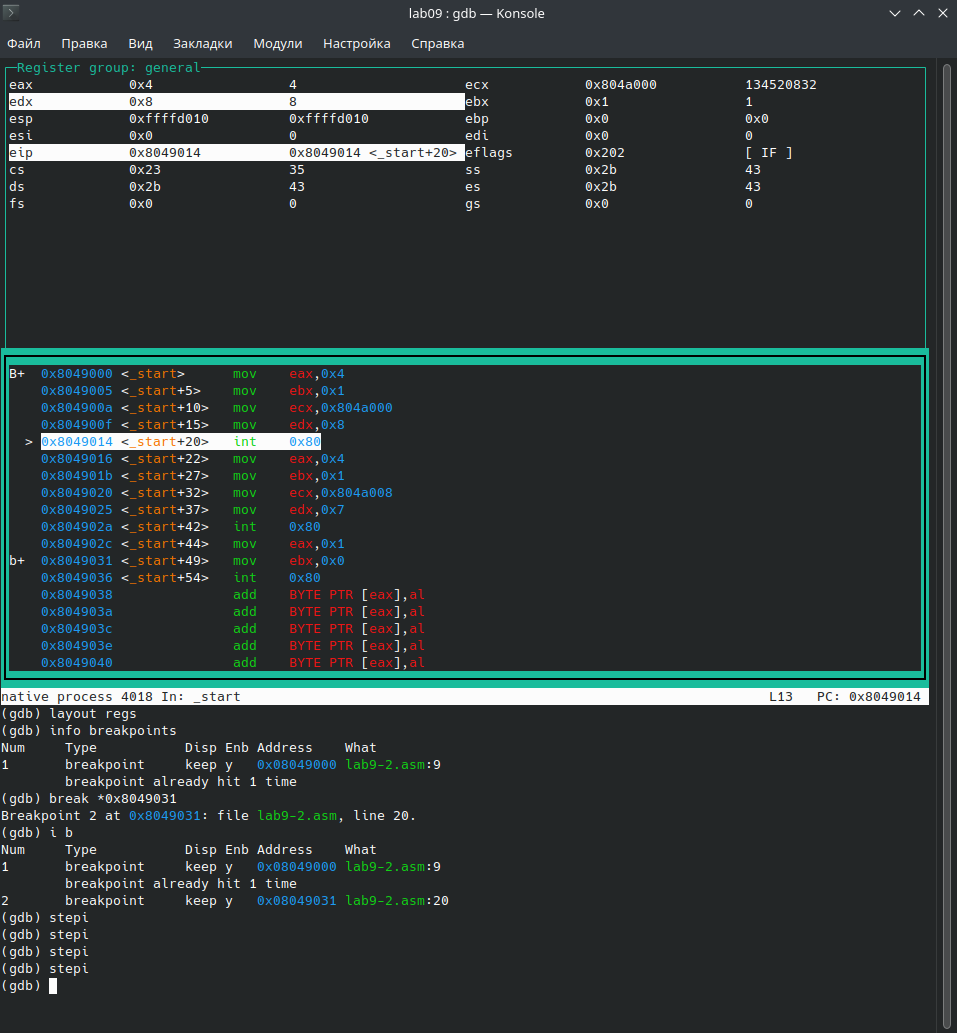
Команда ‘stepi’ 1



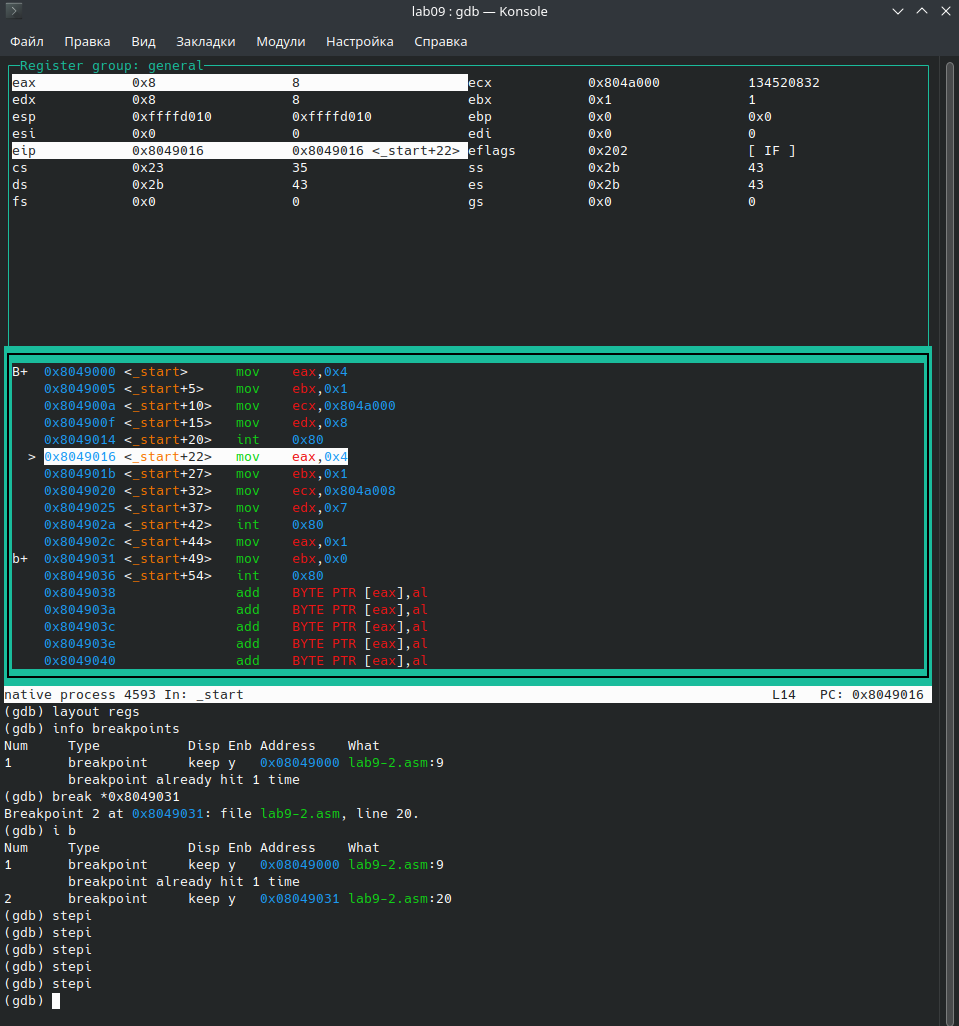
Команда ‘stepi’ 2



Команда ‘stepi’ 3



Команда ‘stepi’ 4

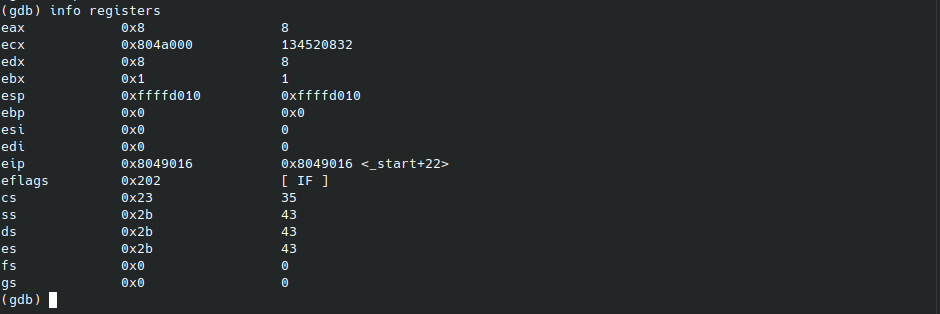


Команда ‘stepi’ 5

Вопрос: значения каких регистров изменяются?

Ответ: во время выполнения команд менялись регистры ebx, ecx, edx, eip и eax.

Смотрим содержимое регистрв с помощью команды info registers’ (рис. [-@fig:025])



Смотрим содержимое регистров

Смотрим значение переменной msg1 по имени (рис. [-@fig:026])

Значение переменной msg1 по имени

Значение переменной msg1 по имени

Смотрим значение переменной msg2 по адресу (рис. [-@fig:027])

Значение переменной msg2 по адресу

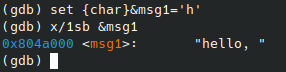
Значение переменной msg2 по адресу

Смотрим инструкцию «mov ecx,msg2», которая записывает в регистр ecx адрес переменной msg2 (рис. [-@fig:028])

Инструкцию «mov ecx,msg2»

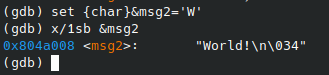
Инструкцию «mov ecx,msg2»

Изменяем первый символ переменной msg1 с помощью команды ‘set’ (большую букву меняем на маленькую) (рис. [-@fig:029])



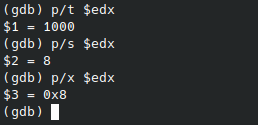
Изменение первого символа переменной msg1

Заменям первый символ во второй переменной msg2 (маленькую букву меняем на большую) (рис. [-@fig:030])



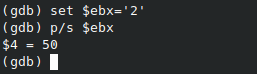
Изменение первого символа переменной ms g2

Смотрим значение регистра edx в различных форматах (сначала в двоичном, потом шестнадцатеричном, а затем в символьном) (рис. [-@fig:031])

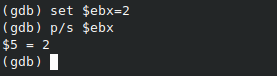


Значение регистра edx в различных форматах

С помощью комнады ‘set’ изменяем значение регистра ebx (рис. [-@fig:032]), (рис. [-@fig:033])



Изменение значение регистра ebx 1

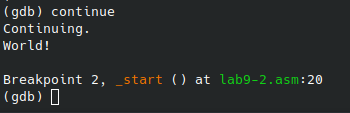


Изменение значение регистра ebx 2

Вопрос: в чём разница вывода команд ‘p/s $ebx‘?

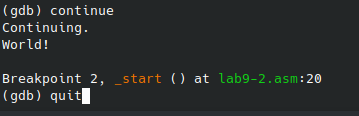
Ответ: Выводятся разные значения, так как команда без кавычек присваивает регистру вводимое значение.

Завершаем выполнение программы с помощью команды ‘continue‘ (рис. [-@fig:034])

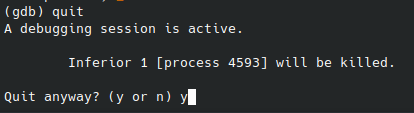


Завершение программы

Выходим из GDB с помощью команды ‘quit‘ (рис. [-@fig:035]), (рис. [-@fig:036])



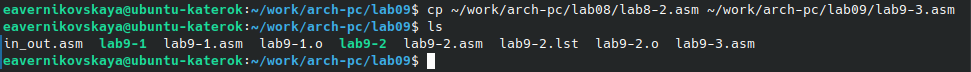
Выход 1



Выход 2

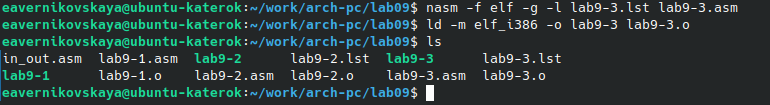
### Обработка аргументов командной строки в GDB

Копируем файл «lab8-2.asm», созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки в файл с именем «lab9-3.asm» (рис. [-@fig:037])



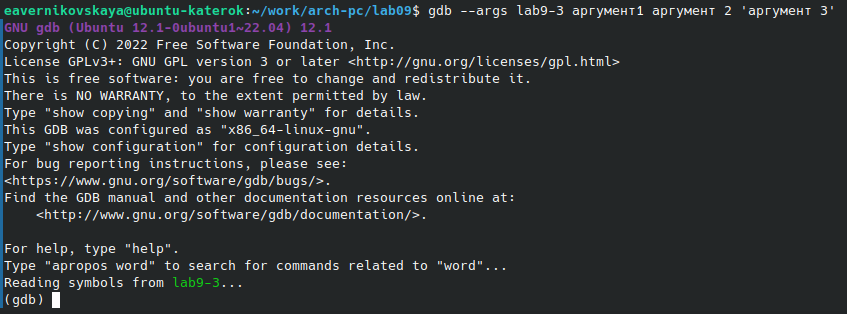
Создание файла «lab9-3.asm»

Создаём исполняемый файл (рис. [-@fig:038])



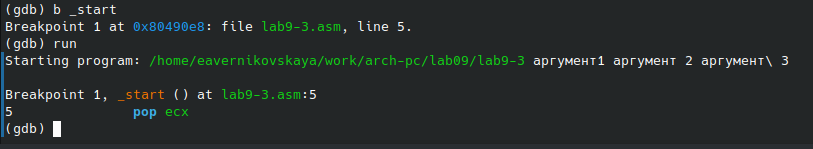
Создание исполняемого файла

Загружаем программу с аргументами в GDB. Для этого используем ключ ‘–args‘ (рис. [-@fig:039])



Загрузка в gdb с использованием ‘–args‘

Устанавливаем точку останова перед первой инструкцией в программе и запускаем её (рис. [-@fig:040])



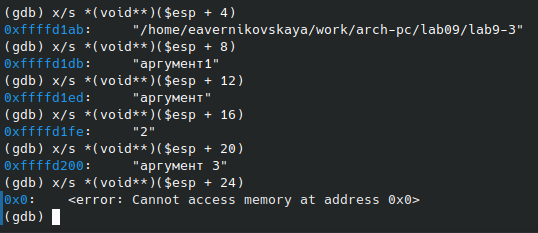
Точка останова + запуск программы

Изучаем адрес вершины стека (рис. [-@fig:041])

Адрес вершины стека

Адрес вершины стека

Смотрим остальные позиции стека (рис. [-@fig:042])



Остальные позиции стека

Вопрос: почему шаг изменения адреса равен 4?

Ответ: шаг изменения адреса равен 4, потому что адресные регистры увеличиваются на 4 при выполнении инструкций в режиме 32-битных (4 байта) процессоров x86. Поэтому при выполнении каждой инструкции адрес следующей инструкции увеличивается на 4. А, например, в случае использованмия режима 64-битных (8 байтаов) процессоров x86 адреса будут увеличиваться на 8.

## Задание для самостоятельной работы

Создаём файл «lab9-4.asm» (рис. [-@fig:043])

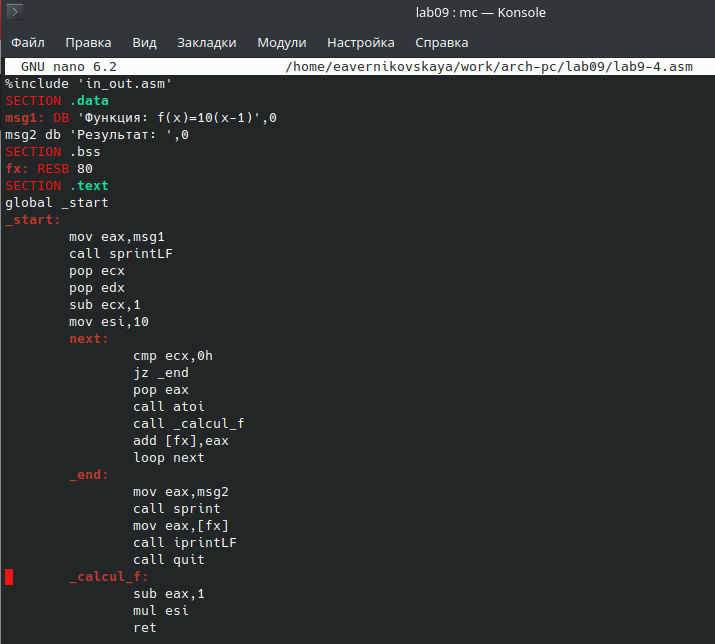
Создание файла «lab9-4.asm»

Создание файла «lab9-4.asm»

Переписываем программу из лабораторной работы №8, реализовав вычисления функции f(x) как подпрограмму (рис. [-@fig:044])

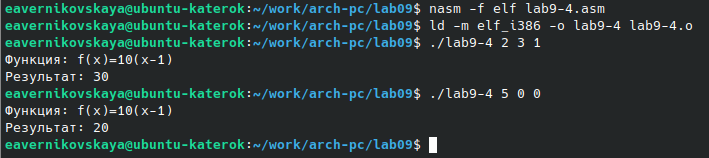
Текст изменённой программы:

%include 'in\_out.asm'  
SECTION .data  
msg1: DB 'Функция: f(x)=10(x-1)',0  
msg2 db 'Результат: ',0  
SECTION .bss  
fx: RESB 80  
SECTION .text  
global \_start  
\_start:  
 mov eax,msg1  
 call sprintLF  
 pop ecx  
 pop edx  
 sub ecx,1  
 mov esi,10  
 next:  
 cmp ecx,0h  
 jz \_end  
 pop eax  
 call atoi  
 call \_calcul\_f  
 add [fx],eax  
 loop next  
 \_end:  
 mov eax,msg2  
 call sprint  
 mov eax,[fx]  
 call iprintLF  
 call quit  
 \_calcul\_f:  
 sub eax,1  
 mul esi  
 ret



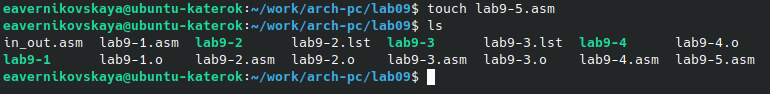
Написание программы

Создаём исполняемый файл и запускаем его (рис. [-@fig:045])



Создание исполняемого файла и его запуск

Создаём файл «lab9-5.asm» (рис. [-@fig:046])

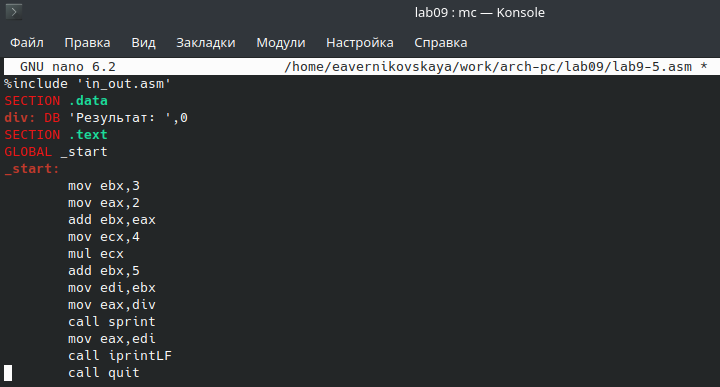


Создание файла «lab9-5.asm»

Вводим текст программы, которая вычисляет выражение (3+2)\*4+5 (рис. [-@fig:047])

Текст программы:

%include 'in\_out.asm'  
SECTION .data  
div: DB 'Результат: ',0  
SECTION .text  
GLOBAL \_start  
\_start:  
 mov ebx,3  
 mov eax,2  
 add ebx,eax  
 mov ecx,4  
 mul ecx  
 add ebx,5  
 mov edi,ebx  
 mov eax,div  
 call sprint  
 mov eax,edi  
 call iprintLF  
 call quit



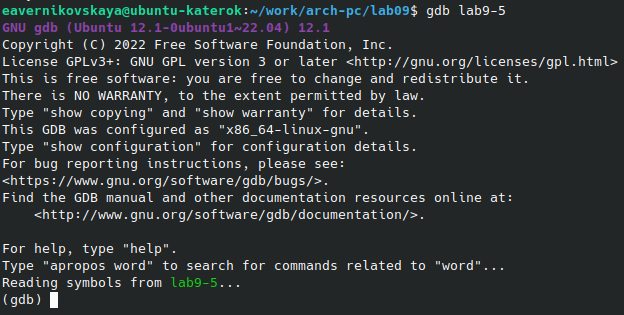
Ввод текста программы

Создаём исполняемый файл (рис. [-@fig:048])

Создание исполняемого файла

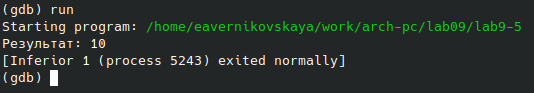
Создание исполняемого файла

Загружаем исполняемый файл в отладчик gdb (рис. [-@fig:049])



Загрузка исполняемого файла в gdb

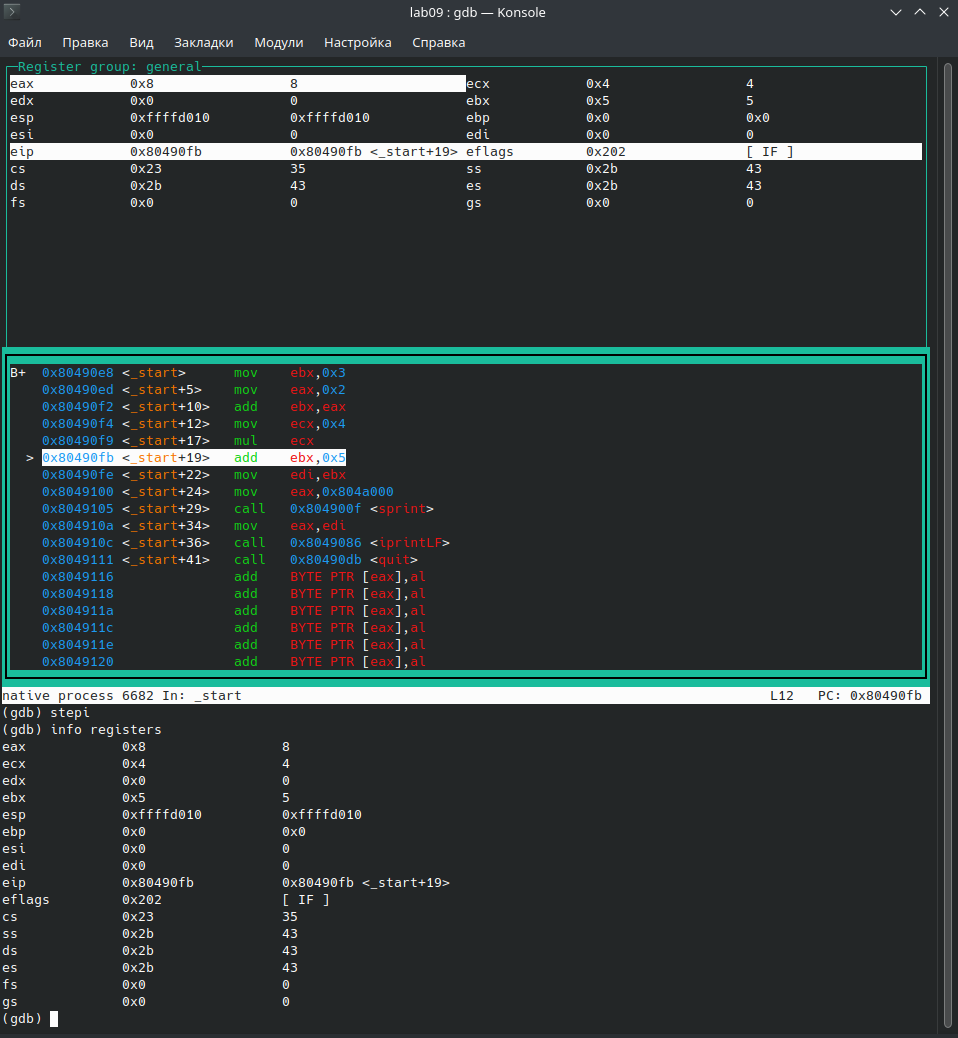
Проверяем работу программы, запустив её в оболочке GDB с помощью команды ‘run’ (рис. [-@fig:050])



Проверка работы программы в оболочке gdb

При запуске программа даёт неверный результат.

Запускаем исполняемый файл в отладчике GDB и смотрим на изменения регистров с помощью команды ‘stepi’ (рис. [-@fig:051])

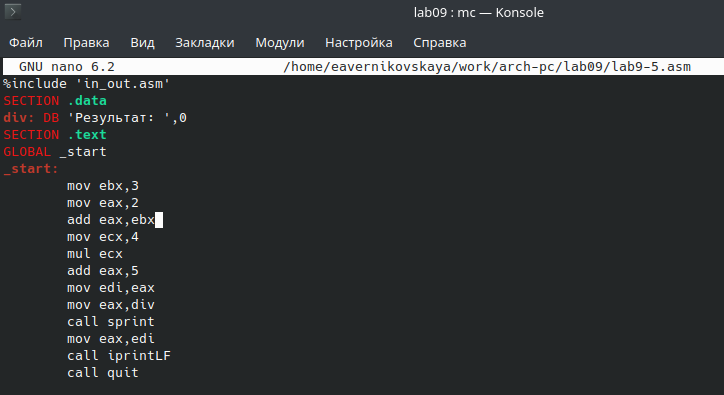


Ищем ошибку регистров

Обнаружив ошибку, исправляем программу (рис. [-@fig:052])

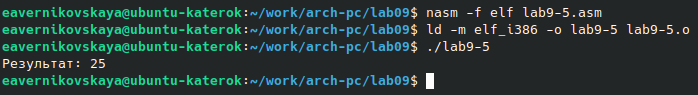
Текст исправленной программы:

%include 'in\_out.asm'  
SECTION .data  
div: DB 'Результат: ',0  
SECTION .text  
GLOBAL \_start  
\_start:  
 mov ebx,3  
 mov eax,2  
 add eax,ebx  
 mov ecx,4  
 mul ecx  
 add eax,5  
 mov edi,eax  
 mov eax,div  
 call sprint  
 mov eax,edi  
 call iprintLF  
 call quit



Исправленная программа

Проверяем работу исправленной программы (рис. [-@fig:053])



Проверка программы

# Выводы

В ходе выполнения лабораторной работы мы приобрели навыки написания программ с использованием подпрограмм. Также мы познакомились с методами отладки при помощи GDB и его основными возможностями.