Отчёт по лабораторной работе №9

Понятие подпрограммы. Отладчик GDB.

Карпухин Клим

Содержание

Список иллюстраций

Список таблиц

1 Цель работы

Цель данной лабораторной работы — приобрести навыки написания программ с использованием подпрограмм на языке ассемблера NASM, а также познакомиться с методами отладки программ с помощью отладчика GDB и его основными возможностями.

2 Выполнение лабораторной работы

2.1 Реализация подпрограмм в NASM

Создал каталог для выполнения лабораторной работы №9, перешёл в него и создал файл `lab09-1.asm` (рис. 2.1):

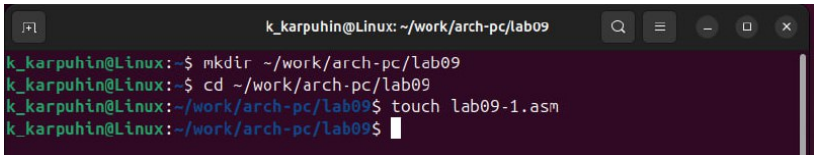

Рисунок 2.1: Создание каталога для программ лабораторной работы №9, переход в него и создание файла lab09-1.asm

2.1.1 Программа вычисления выражения $f(x) = 2x + 7$ с использованием подпрограммы

Ввёл в файл `lab09-1.asm` текст программы из листинга 9.1, реализующей вычисление выражения $f(x) = 2x + 7$ с помощью подпрограммы `_calcul` (рис. 2.2).

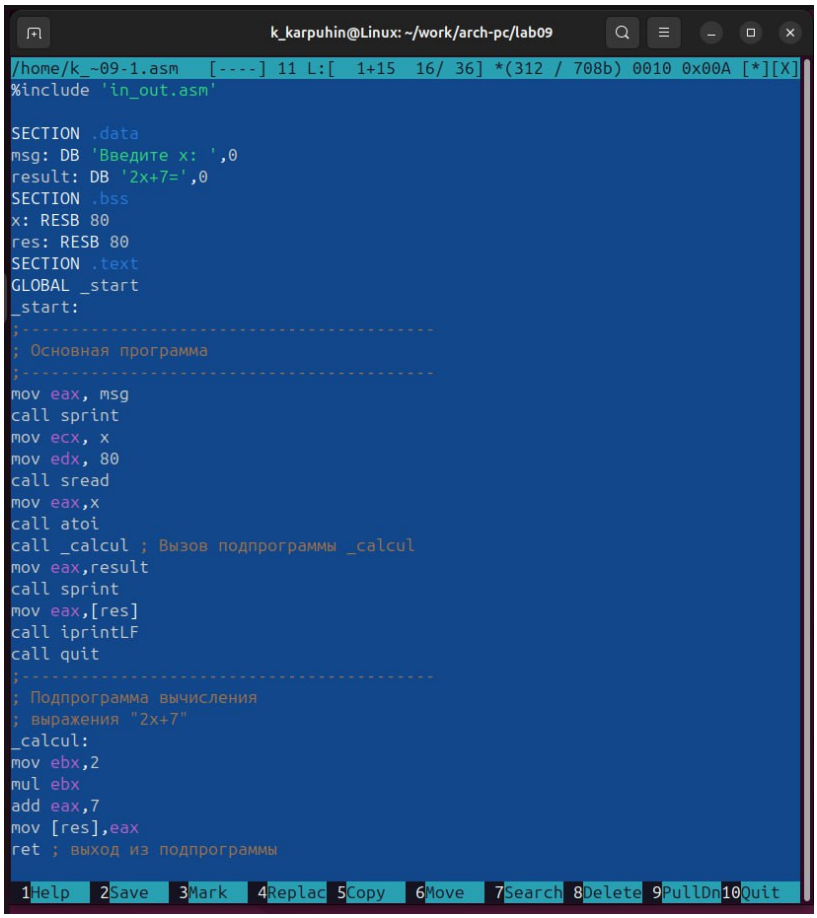

Рисунок 2.2: Текст программы lab09-1.asm из листинга 9.1

Создал объектный и исполняемый файлы и проверил работу программы (рис. 2.3).

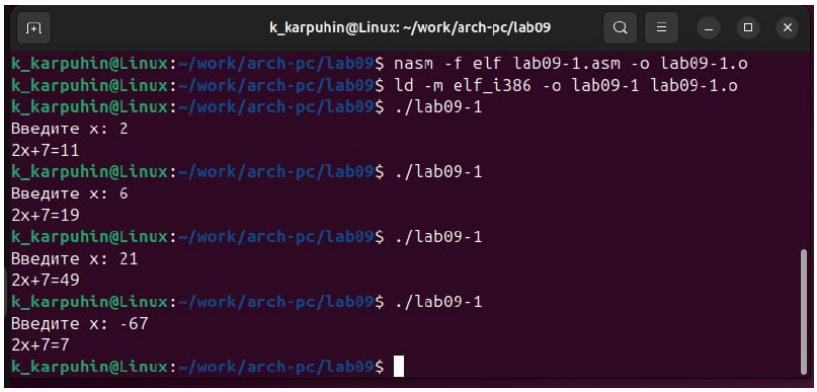

Рисунок 2.3: Трансляция, компоновка и запуск программы lab09-1.asm

При вводе различных **неотрицательных** значений x (например, x = 2, x =

6 и x = 21) программа корректно вычисляла значение выражения 2x + 7 и выводила результат на экран. При вводе отрицательных значений (например, x = -67) результат некорректен из-за того, что стандартная процедура atoi из файла in_out.asm не обрабатывает знак минус и предназначена только для преобразования неотрицательных целых чисел.

2.1.2 Модификация программы: вычисление $f(g(x))$

По заданию изменил текст программы, добавив подпрограмму _subcalcul в подпрограмму _calcul для вычисления выражения

$$f(g(x)), \quad f(x) = 2x + 7, \quad g(x) = 3x - 1.$$

Логика работы: 1. Пользователь вводит x 2. x передаётся в _calcul 3. _calcul вызывает _subcalcul с x 4. _subcalcul вычисляет $3x - 1$ и возвращает результат 5. _calcul вычисляет $2 \cdot (3x - 1) + 7$ 6. Результат выводится на экран

Модифицированный фрагмент программы имеет вид (рис. 2.4):

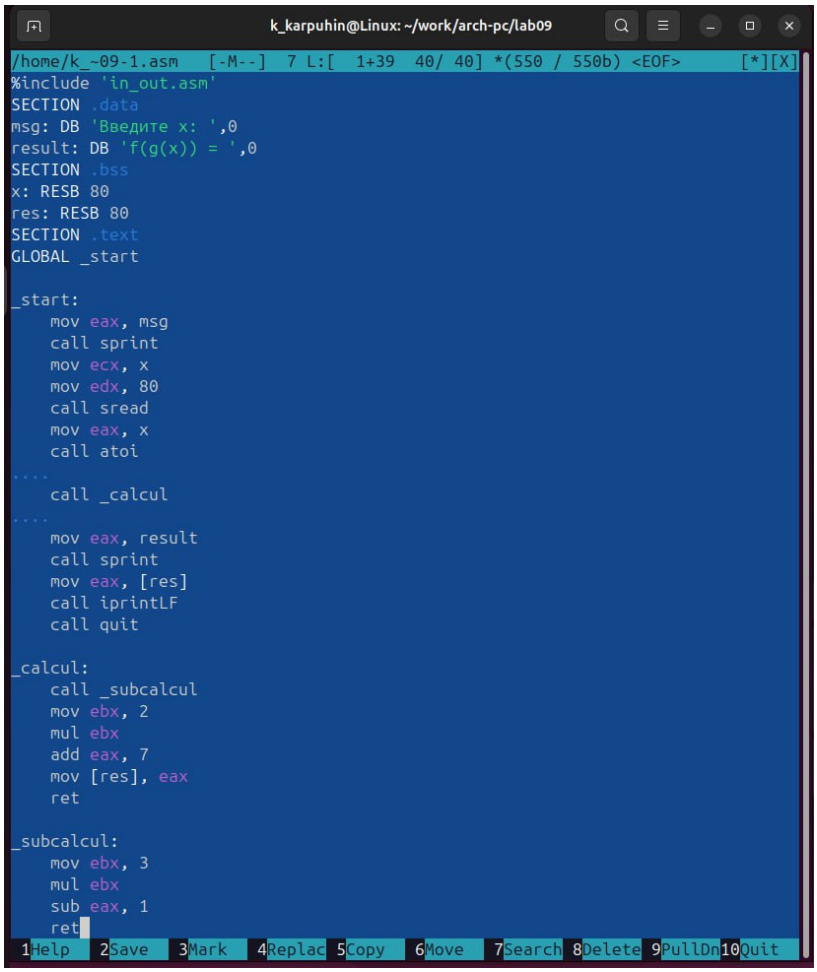

Рисунок 2.4: Изменённый текст программы lab09-1.asm с подпрограммами _calcul и _subcalcul

После пересборки программы и запуска с несколькими тестовыми значениями x убедился, что программа корректно вычисляет $f(g(x))$ (рис. 2.5).

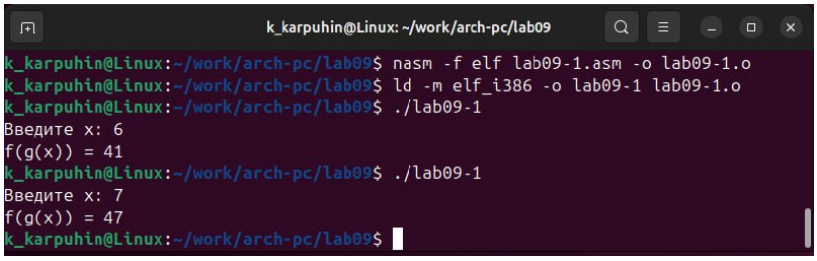

Рисунок 2.5: Запуск модифицированной программы и проверка корректности вычисления f(g(x))

2.2 Отладка программ с помощью GDB

2.2.1 Программа вывода сообщения «Hello, world!»

Создал файл `lab09-2.asm` с текстом программы из листинга 9.2, выводящей сообщение «Hello, world!» (рис. 2.6).

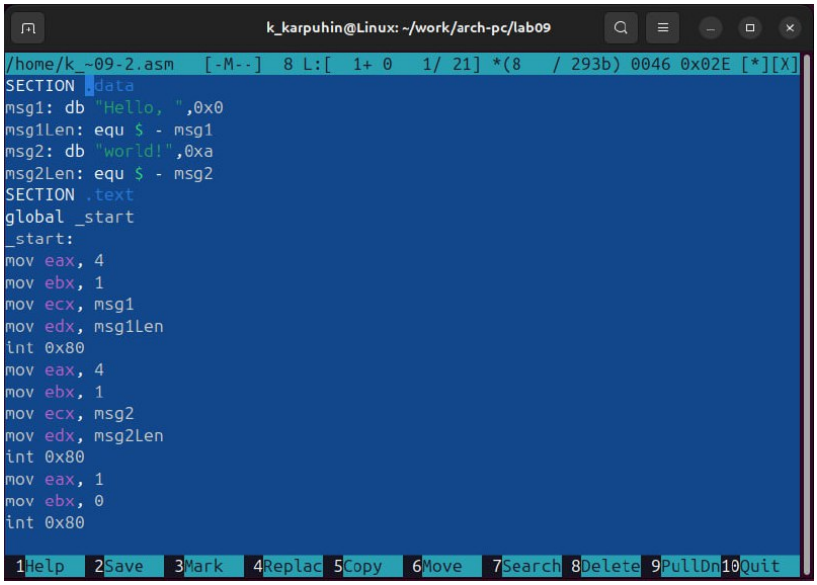

Рисунок 2.6: Текст программы lab09-2.asm — вывод сообщения Hello, world!

Собрал программу с добавлением отладочной информации (`-g`) и получил исполняемый файл (рис. 2.7).

```
nasm -f elf -g -l lab09-2.lst lab09-2.asm
ld -m elf_i386 -o lab09-2 lab09-2.o
```

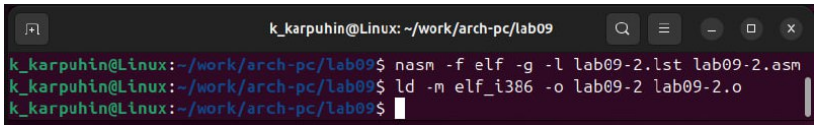

Рисунок 2.7: Трансляция и компоновка программы lab09-2 с ключом -g

2.2.2 Запуск программы в GDB и установка точки останова

Загрузил исполняемый файл в отладчик GDB. Внутри GDB запустил программу командой `run` и убедился, что она корректно выводит строку (рис. 2.8).

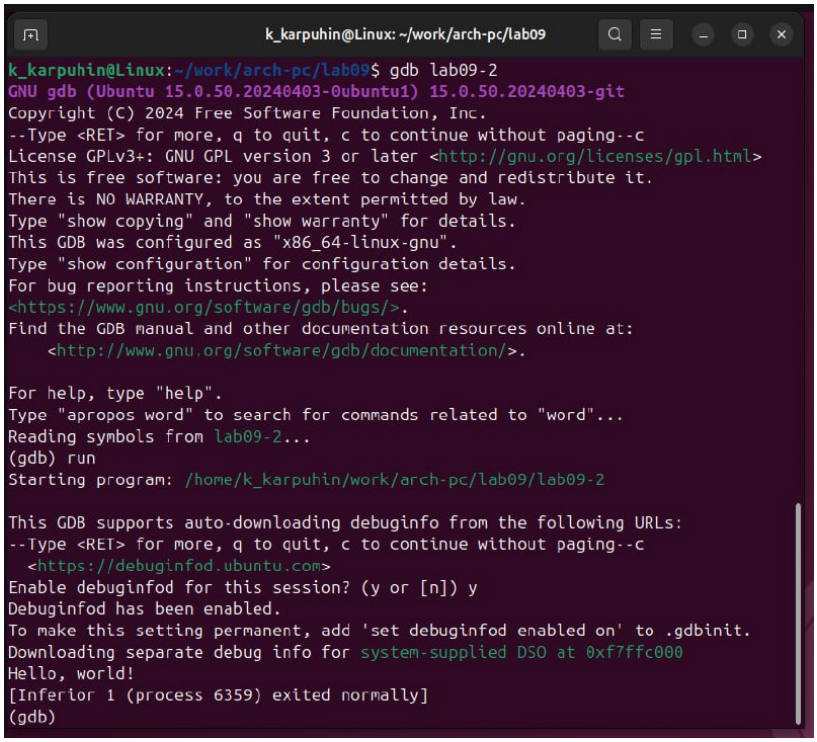

Рисунок 2.8: Запуск программы lab09-2 внутри GDB

Далее установил точку останова на метку `_start` и снова запустил программу (рис. 2.9).

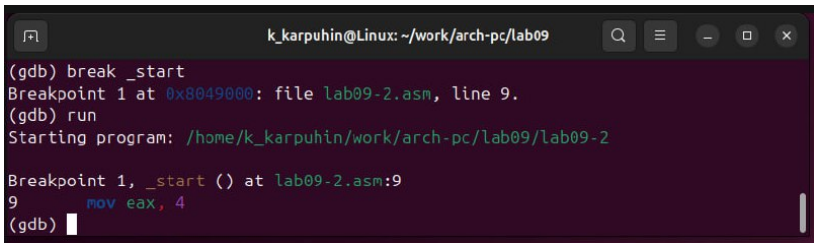

Рисунок 2.9: Установка точки останова на метку _start и запуск программы до точки останова

2.2.3 Дизассемблирование и режим Intel-синтаксиса

Посмотрел дизассемблированный код программы, начиная с метки `_start`, затем переключил отображение команд на Intel-синтаксис.

В режиме ATT и Intel различается порядок операндов и общее оформление инструкций. В режиме Intel синтаксис совпадает с используемым в NASM, что упрощает сопоставление дизассемблированного кода с исходником (рис. 2.10).

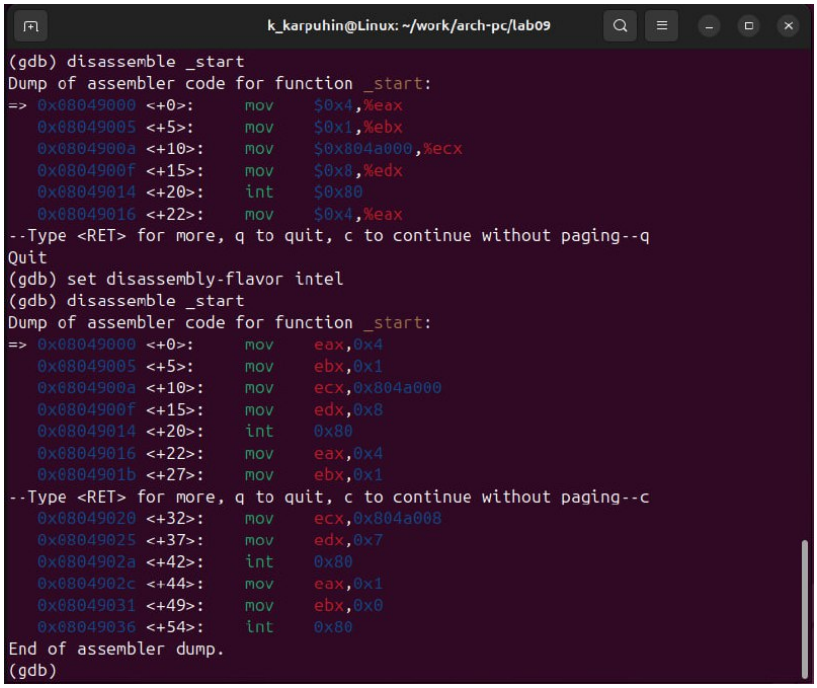

Рисунок 2.10: Дизассемблирование программы lab09-2 в режимах ATT и Intel

2.2.4 Режим псевдографики GDB

Для более удобного анализа программы включил псевдографический режим (рис. 2.11).

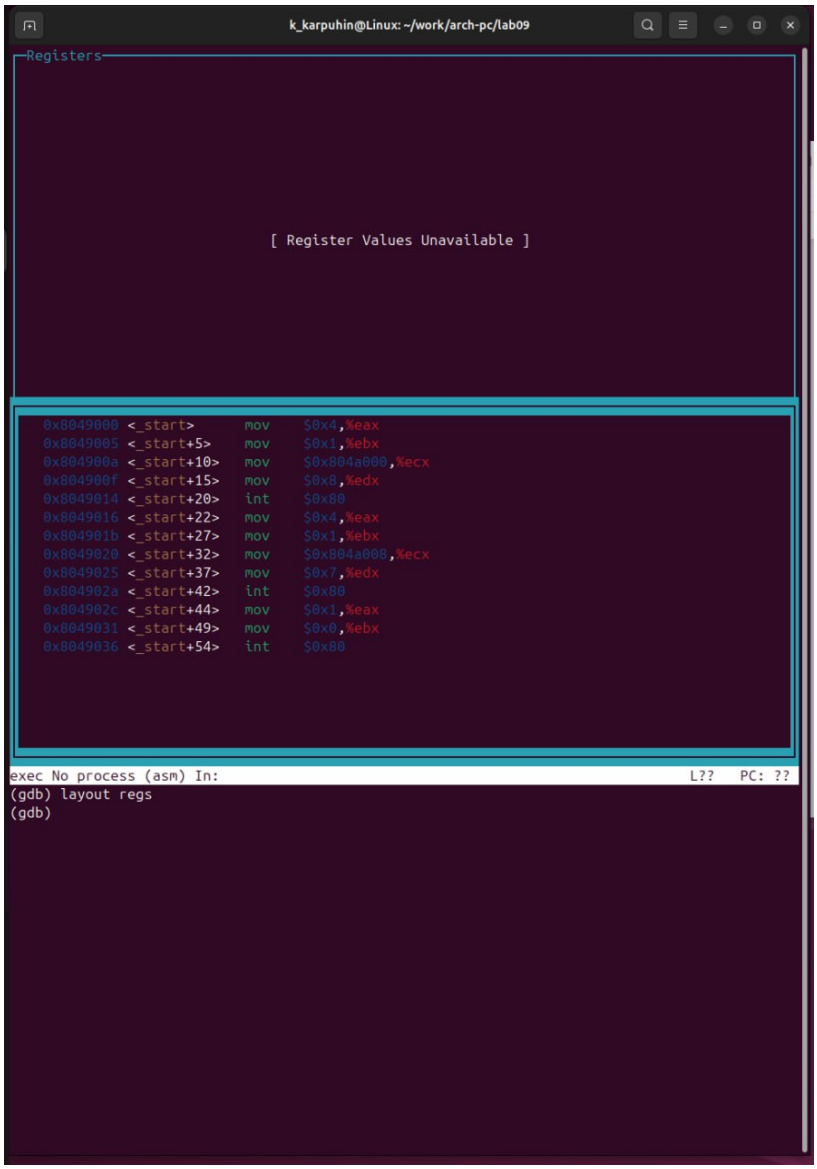

Рисунок 2.11: Режим псевдографики GDB (layout asm и layout regs)

2.2.5 Работа с точками останова

Проверил список установленных точек останова, затем по заданию установил ещё одну точку останова по адресу предпоследней инструкции. После этого ещё раз вывел информацию о всех точках останова (рис. 2.12).

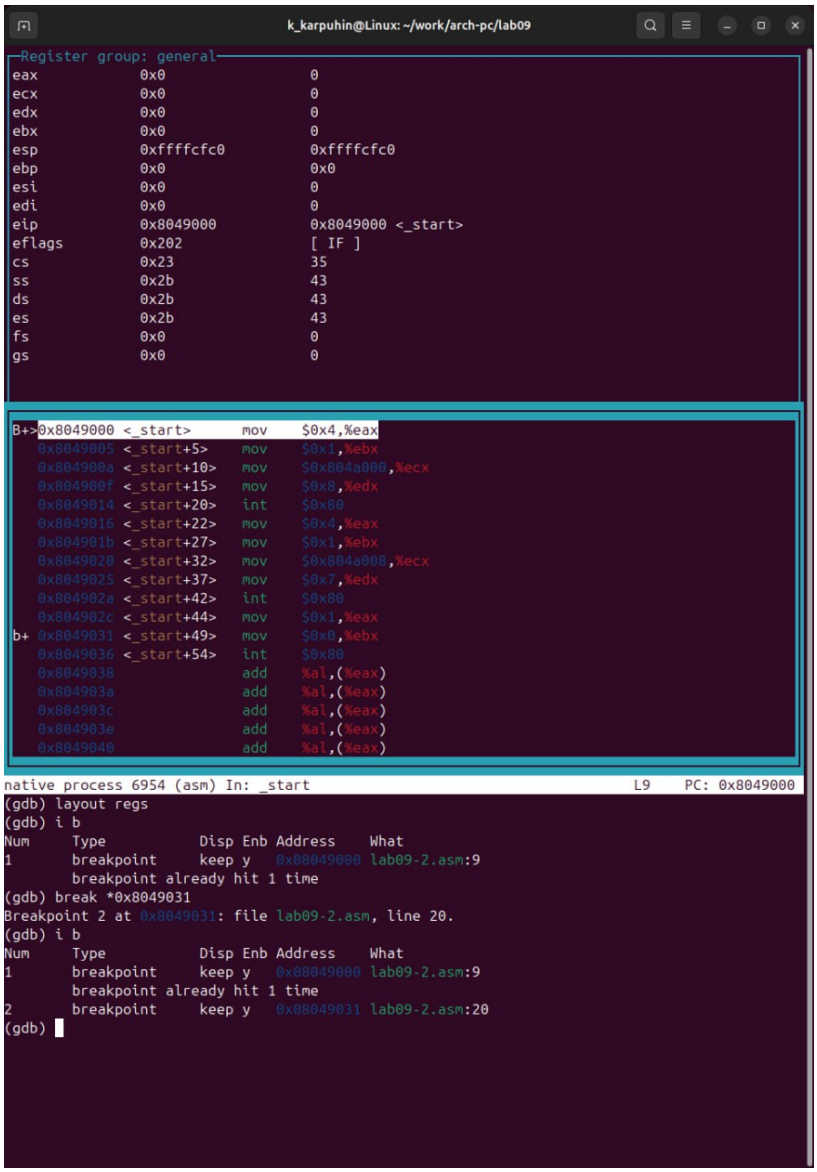

Рисунок 2.12: Список точек останова и установка брейкпоинта по адресу

2.2.6 Пошаговое выполнение и анализ регистров

С помощью команды `stepi` последовательно выполнил первые пять инструкций функции `_start` (`mov eax,4`, `mov ebx,1`, `mov ecx`,msg1, `mov edx,msg1Len`, `int 0x80`) и по окну регистров (`layout regs`) проследил, что изменяются значения регистров `eax`, `ebx`, `ecx`, `edx`, а также счётчика команд

`eip`.

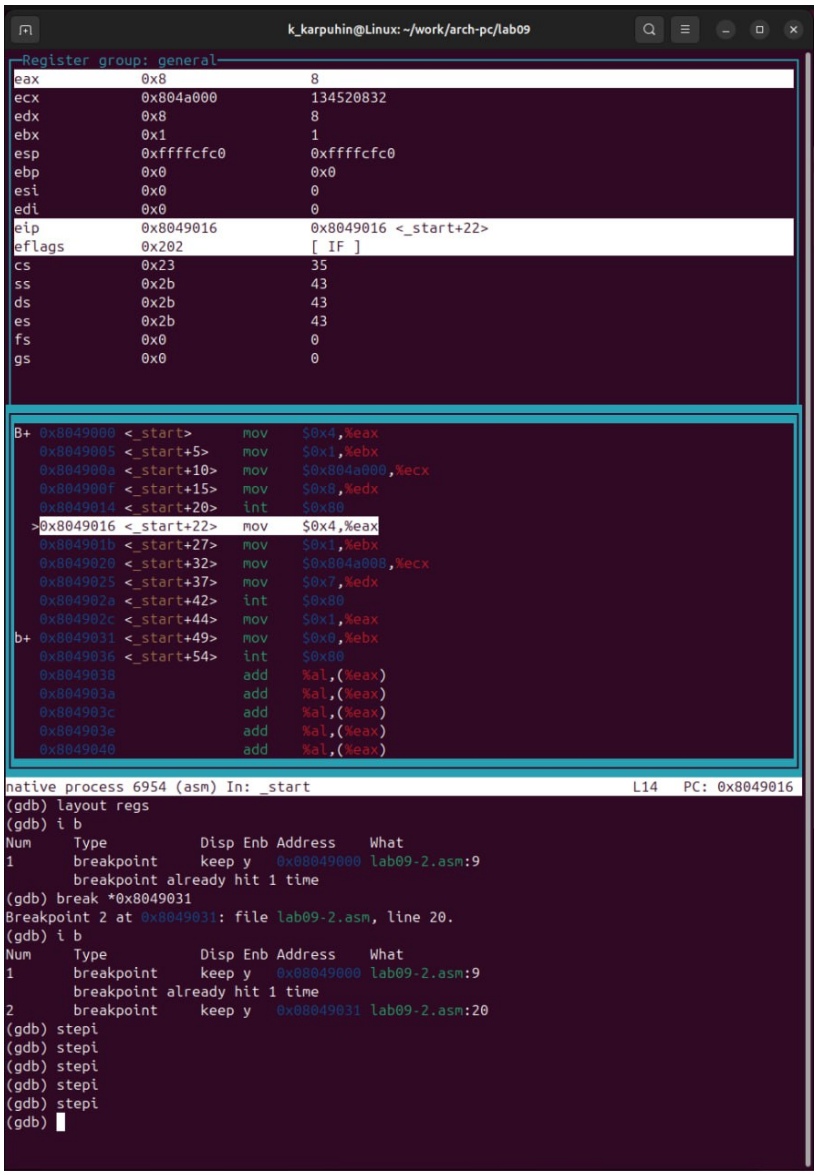

Рисунок 2.13: Пошаговое выполнение программы и анализ изменений регистров

2.2.7 Работа с данными программы в GDB

Для просмотра содержимого памяти использовал команду `x` в различных форматах. По имени переменной `msg1` просмотрел её содержимое. В ответ отладчик показал строку `"Hello, "`.

Аналогично, по адресу, записанному в регистре `ecx` после инструкции `mov ecx, msg2`, просмотрел содержимое переменной `msg2` (рис. 2.14).

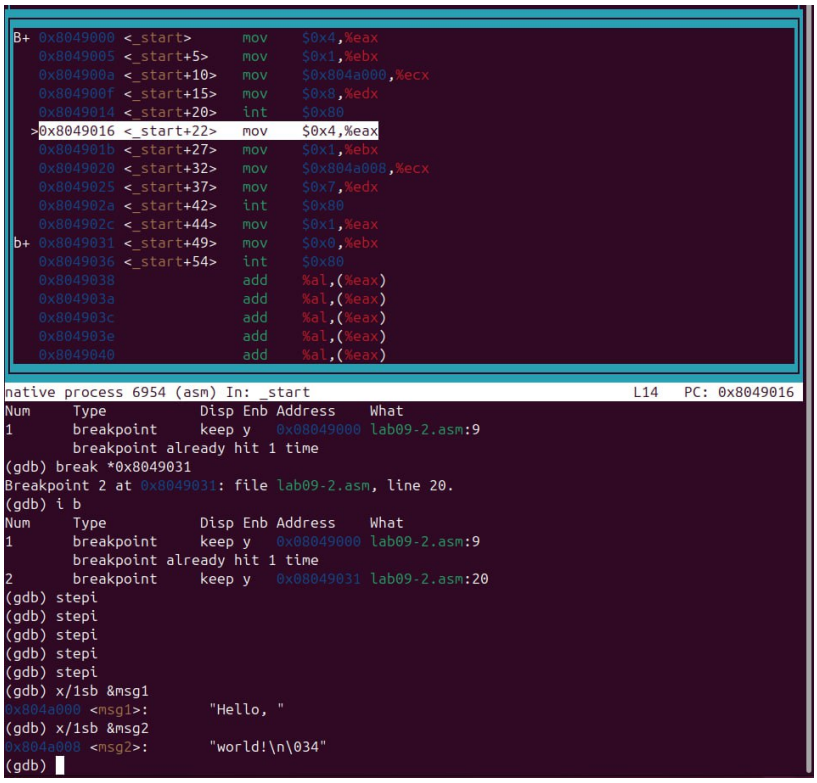

Рисунок 2.14: Использование команды x для просмотра содержимого переменных msg1 и msg2

Изменил первый символ переменной `msg1` с помощью команды `set`. Аналогично изменил первый символов в переменной `msg2` (рис. 2.15).

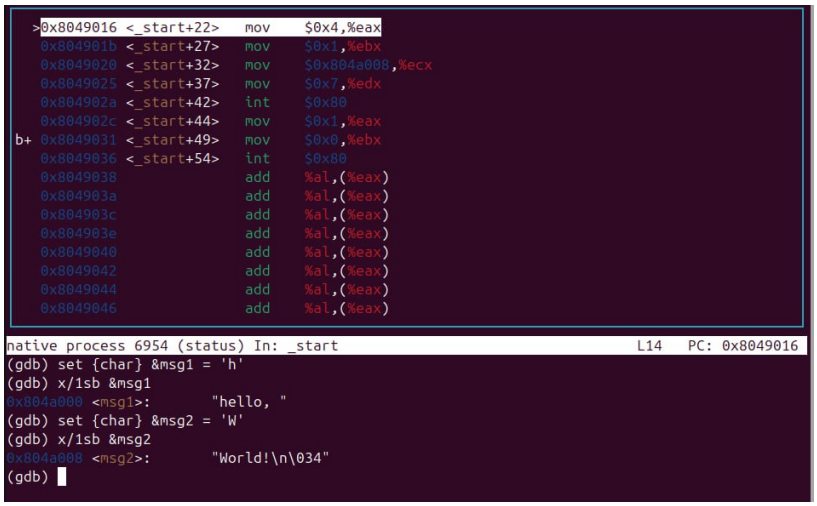

Рисунок 2.15: Изменение содержимого строковых переменных с помощью команды set

Вывел в различных форматах значение регистра `edx`, затем изменил значение регистра `ebx` (рис. 2.16).

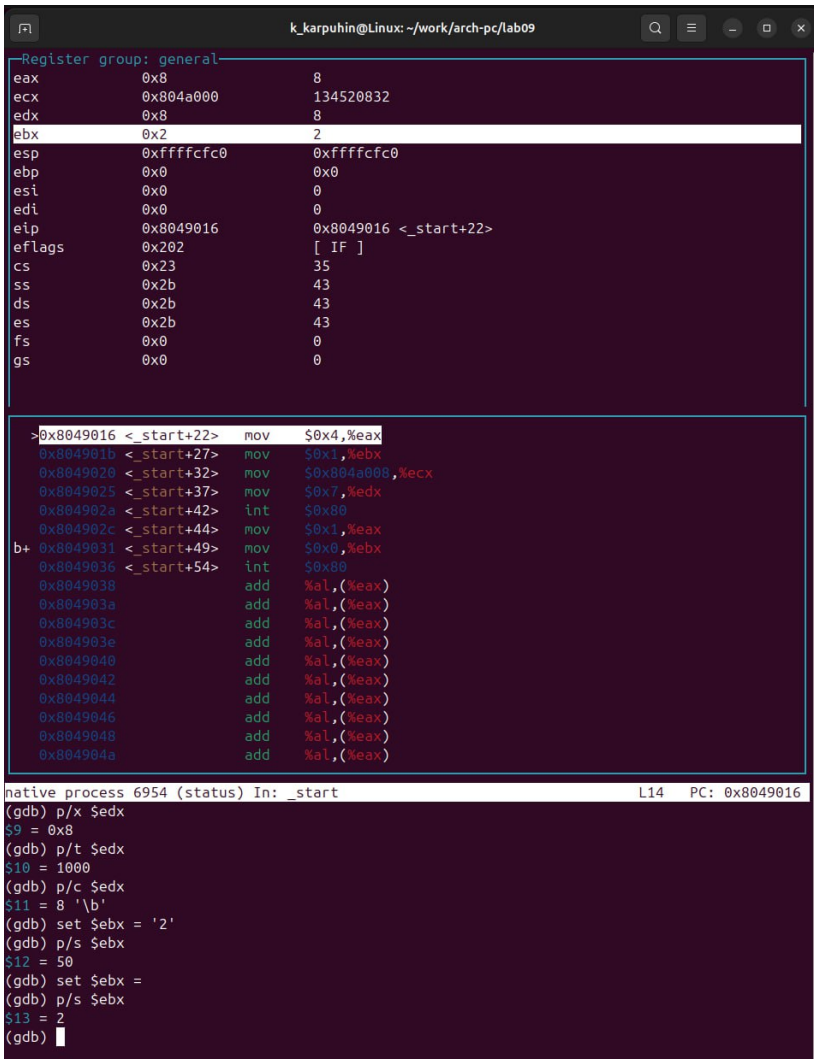

Рисунок 2.16: Примеры использования команды print и изменения регистра ebx

Разница в выводе `p/s $ebx` связана с тем, что в первом случае в регистр `ebx` записан символьный код `'2'`, и отладчик интерпретирует его как адрес строки, а во втором случае — числовое значение 2.

Завершил выполнение программы командой `continue` (c), а затем вышел из GDB (рис. 2.17).

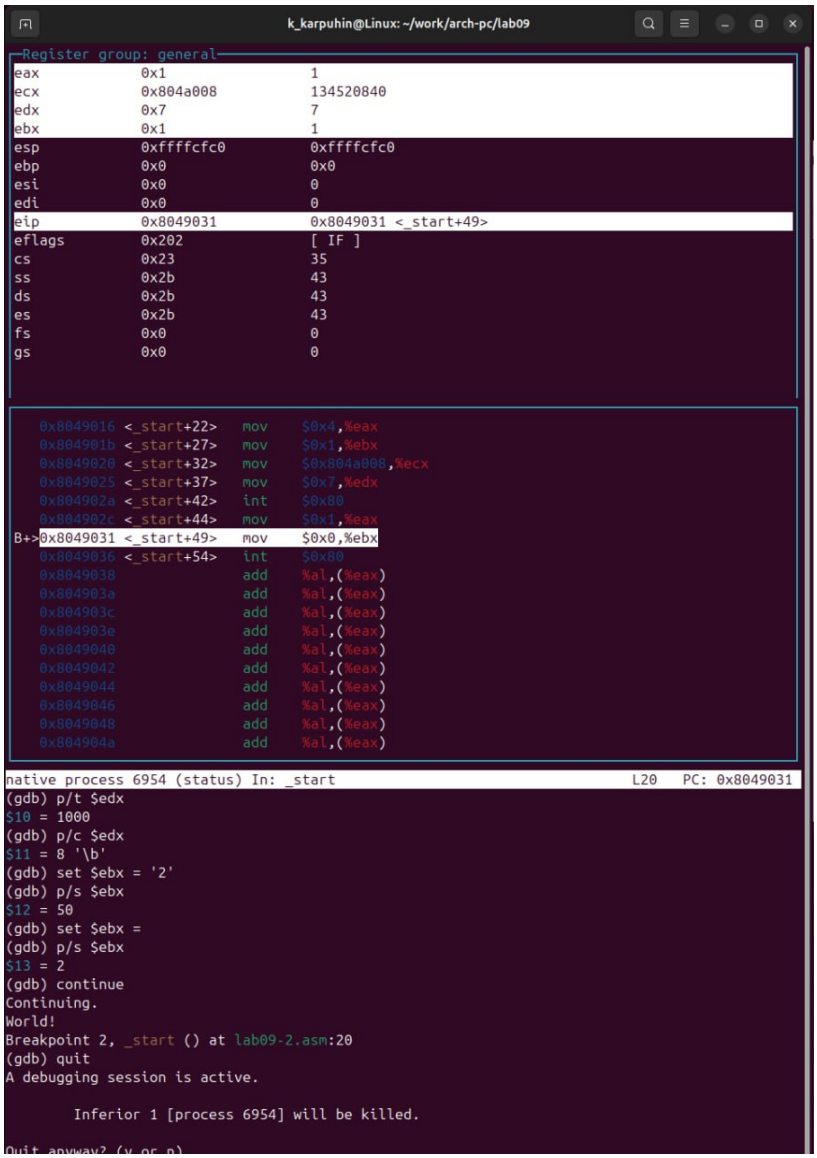

Рисунок 2.17: Завершение выполнения программы и выход из GDB

2.3 Обработка аргументов командной строки в GDB

Скопировал файл программы вывода аргументов командной строки из лабораторной работы №8, создал объектный и исполняемый файлы с отладочной информацией (рис. 2.18).

Рисунок 2.18: Копирование файла программы вывода аргументов командной строки из лабораторной работы №8, создание объектного и исполняемого файлов

Загрузил программу в GDB с использованием ключа `--args`, указав аргументы (рис. 2.19).

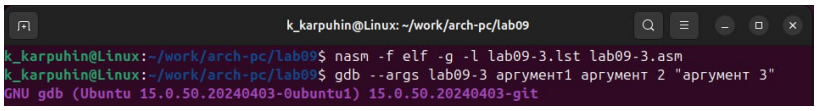

Рисунок 2.19: Загрузка программы lab09-3 в GDB с аргументами командной строки

Установил точку останова на метку `_start` и запустил программу (рис. 2.20).

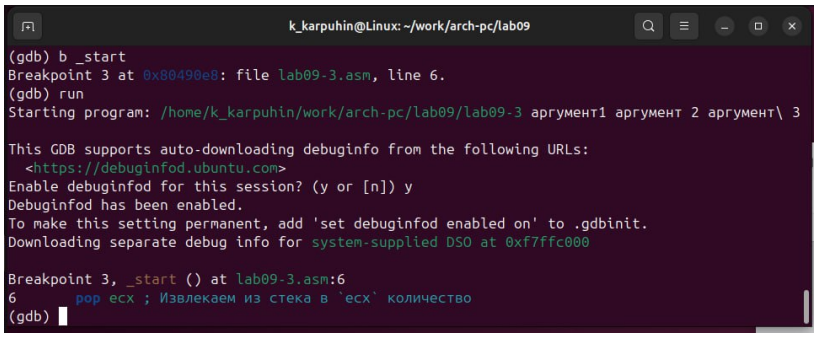

Рисунок 2.20: Установка точки останова и запуск программы

Значение вершины стека хранится в регистре `esp`. Просмотрел содержимое по адресу `esp`. На вершине стека хранилось число `0x5`, то есть общее количество аргументов: lab09-3, аргумент1, аргумент, 2, аргумент 3.

Далее просмотрел остальные элементы стека, интерпретируя их как адреса строк (рис. 2.21).

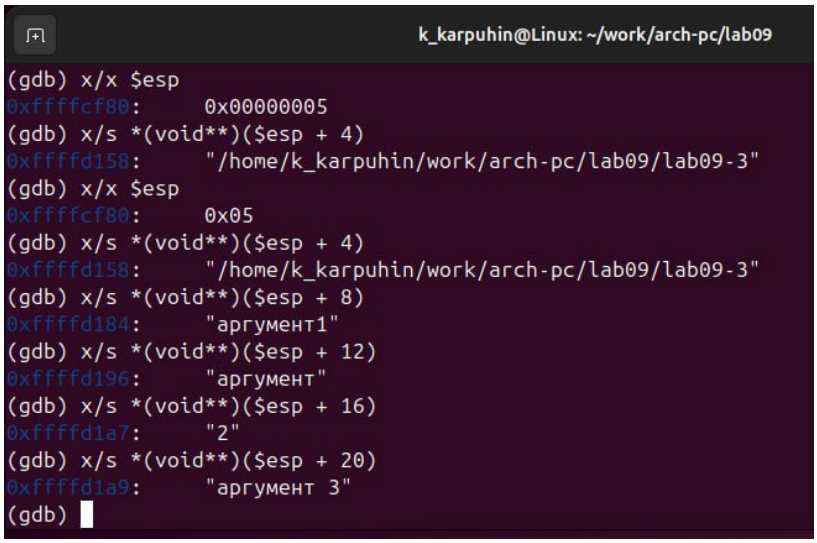

Рисунок 2.21: Просмотр расположения аргументов командной строки в стеке

Шаг изменения адреса равен 4 байтам (`[esp+4]`, `[esp+8]`, `[esp+12]` и т.д.), так как в стеке хранятся указатели (адреса строк), каждый из которых занимает 4 байта в 32-разрядной архитектуре.

3 Задание для самостоятельной работы

3.1 Задание 1. Преобразование программы ЛР8 с использованием подпрограммы

В задании требуется преобразовать программу из лабораторной работы №8 (задание №1 для самостоятельной работы), реализуя вычисление значения функции $f(x)$ как подпрограмму.

В ЛР8 была реализована программа, вычисляющая сумму

$$f(x_1) + f(x_2) + \cdots + f(x_n),$$

где значения x_i передаются через аргументы командной строки.

Для своего варианта (вариант 1) использую функцию:

$$f(x) = 2x + 15$$

Создал файл `lab09-task1.asm` и реализовал следующую структуру программы:

- основная программа:
  - извлекает из стека количество аргументов и имя программы;

- в цикле читает очередной аргумент x_i в виде строки;
- преобразует строку в число (процедура `atoi`);
- вызывает подпрограмму `_func` для вычисления $f(x_i)$;
- накапливает сумму значений функции;
- по завершении цикла выводит результат;

- подпрограмма `_func` реализует вычисление $f(x) = 7 + 2x$.

Текст программы приведён ниже (рис. 3.1):

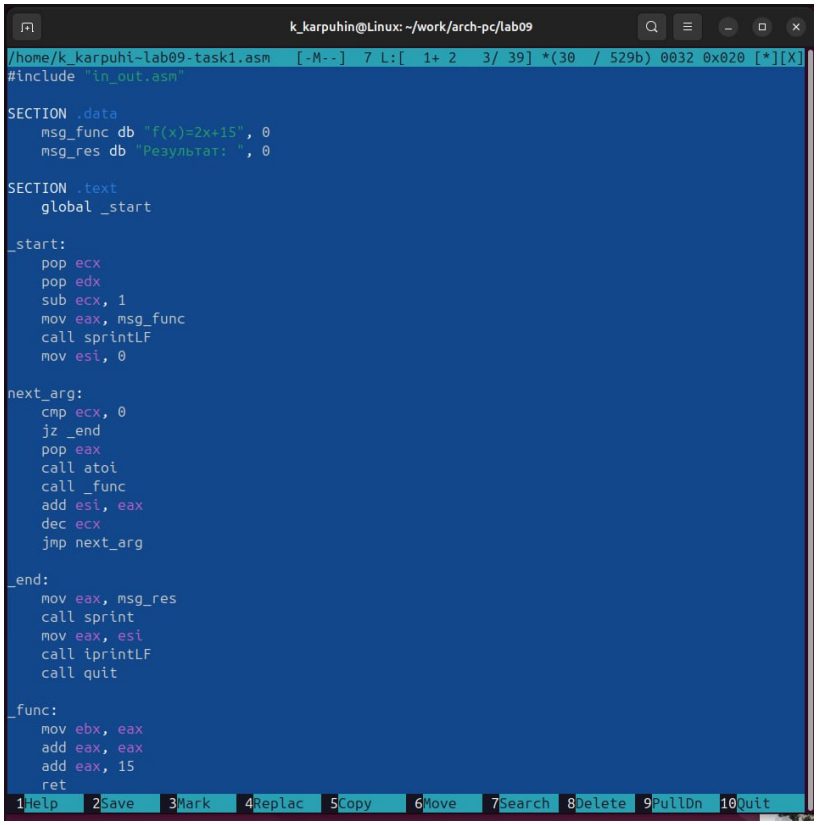

Рисунок 3.1: Текст программы task1.asm для вычисления суммы значений функции f(x) как подпрограммы

Собрал и запустил программу с набором аргументов 1 2 3 4 (рис. 3.2).

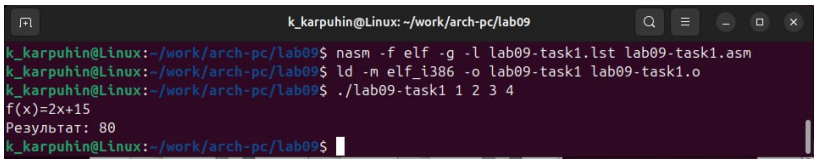

Рисунок 3.2: Запуск программы task1 с набором аргументов 1 2 3 4

В этом случае:

- $f(1) = 15 + 2 \cdot 1 = 17$
- $f(2) = 15 + 2 \cdot 2 = 19$
- $f(3) = 15 + 2 \cdot 3 = 21$
- $f(4) = 15 + 2 \cdot 4 = 23$

Сумма:

$$17 + 19 + 21 + 23 = 80.$$

Программа вывела результат 80, что подтверждает корректность реализации подпрограммы `_func` и всей программы в целом.

3.2 Задание 2. Поиск и исправление ошибки в программе вычисления выражения

Скопировал программу из листинга 9.3 (рис. 3.3).

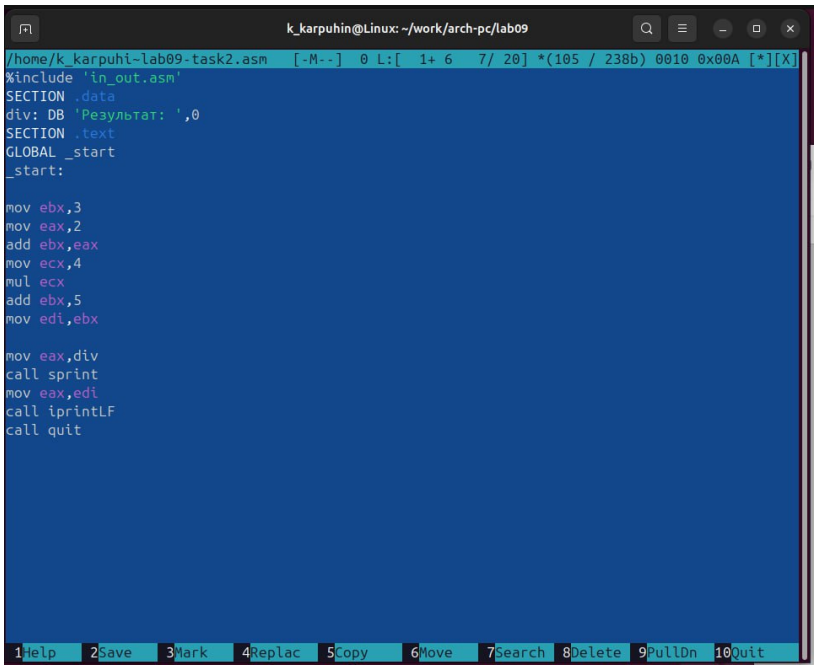

Рисунок 3.3: Текст программы из листинга 9.3

Собрал и запустил программу из листинга 9.3 (рис. 3.4).

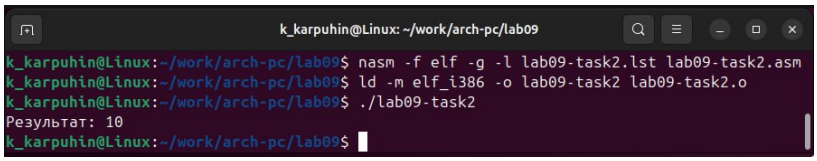

Рисунок 3.4: Запуск программы из листинга 9.3

Ожидаемое значение выражения:

$$(3 + 2) * 4 + 5 = 5 * 4 + 5 = 20 + 5 = 25.$$

Однако программа выводит значение 10.

3.2.1 Поиск ошибки с помощью GDB

Загрузил программу в GDB и установил точку останова на `_start` (рис. 3.5).

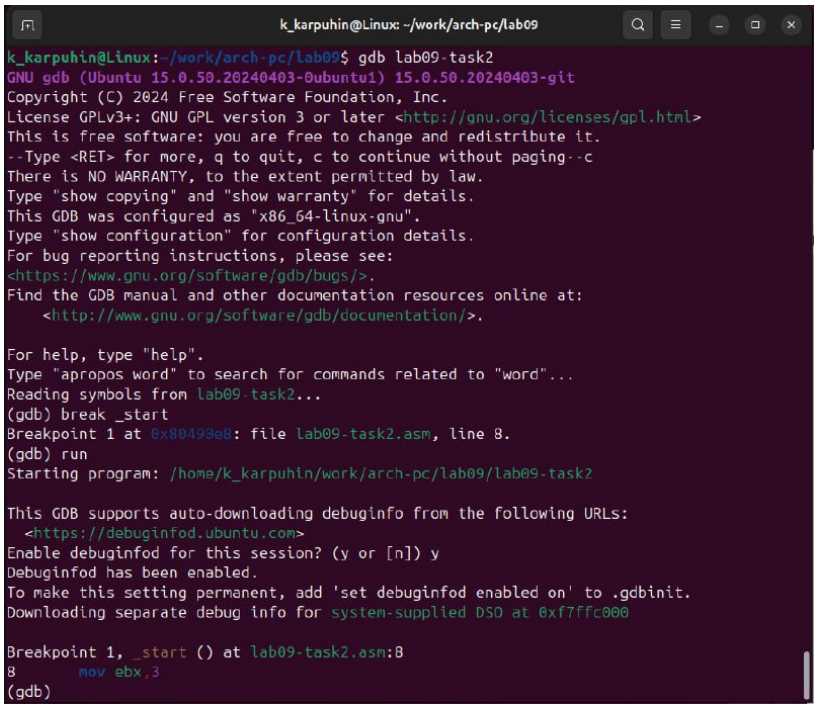

Рисунок 3.5: Загрузка программы в GDB и установка точки останова

Пошагово выполнял программу командой `stepi`, одновременно просматривая значения регистров (рис. 3.6).

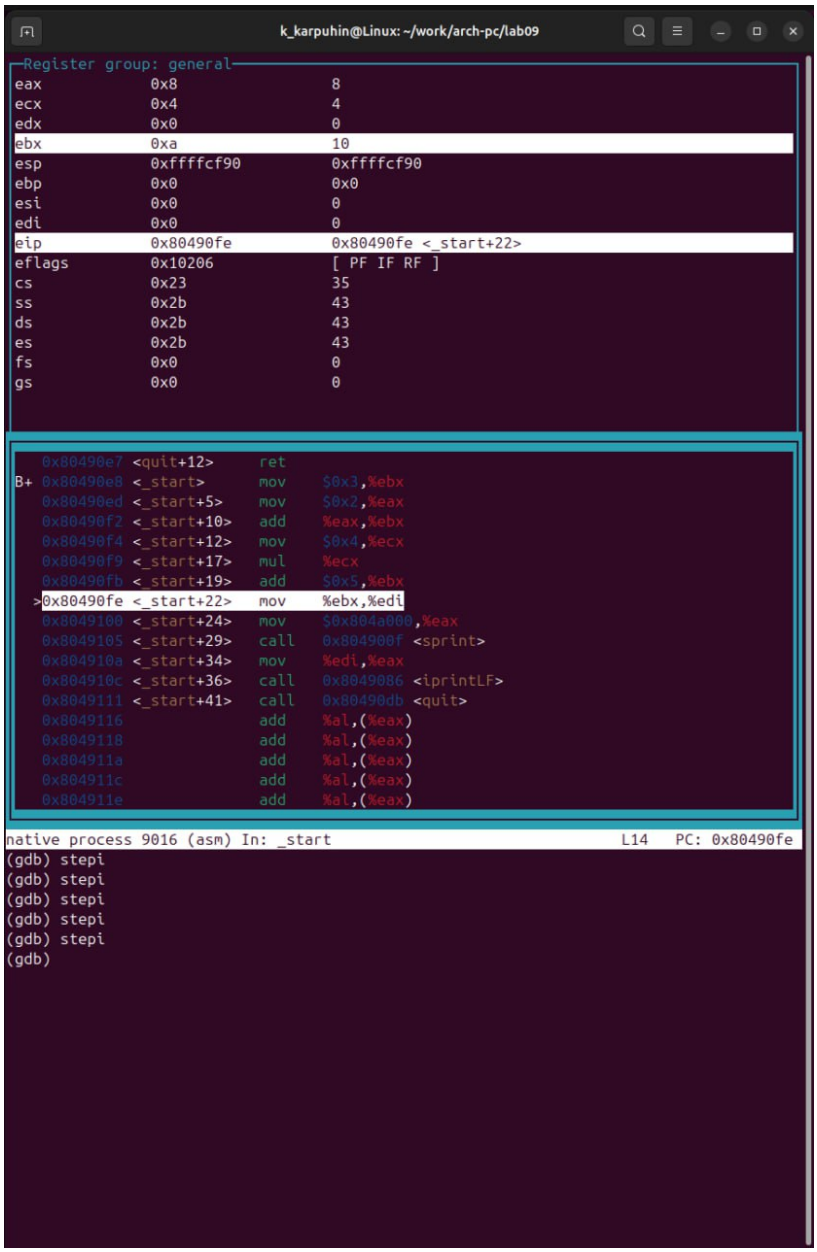

Рисунок 3.6: Пошаговое выполнение программы

После выполнения инструкций:

- `mov ebx,3` → `ebx = 3`
- `mov eax,2` → `eax = 2`
- `add ebx,eax` → `ebx = 5` (сумма $3 + 2$)
- `mov ecx,4` → `ecx = 4`

- `mul ecx` → `eax = 2 * 4 = 8, edx = 0`

Видно, что умножение выполняется над значением `eax = 2`, а не над суммой $3 + 2$, которая хранится в `ebx`. Далее выполняется `add ebx, 5`, и в итоге в `ebx` оказывается значение `10`, которое и печатается на экран.

Ошибка состоит в том, что перед умножением в `eax` не записана сумма $(3 + 2)$.

3.2.2 Исправление программы

Чтобы получить правильное выражение $(3 + 2) * 4 + 5$, нужно умножать именно сумму, а затем добавить 5 (рис. 3.7).

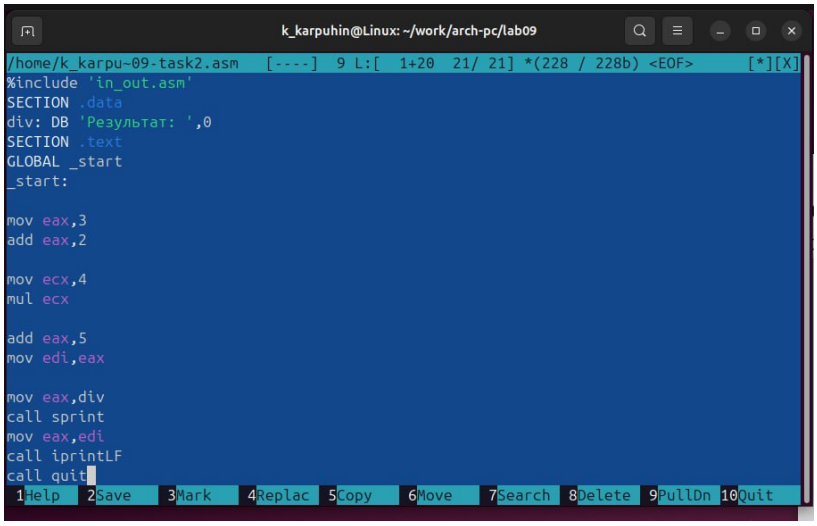

Рисунок 3.7: Исправленный текст программы вычисления выражения (3+2)*4+5

Пересобрал и запустил исправленную программу (рис. 3.8).

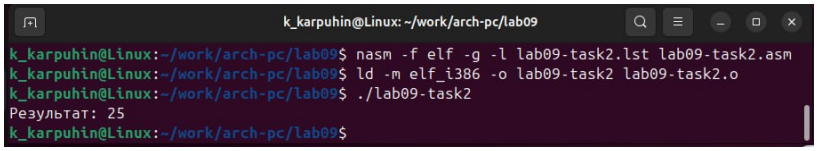

Рисунок 3.8: Запуск исправленной программы и вывод корректного результата

Теперь результат 25, что соответствует ожидаемому значению выражения.

4 Выводы

В ходе выполнения данной лабораторной работы я:

- закрепил понятие подпрограммы и на практике реализовал вызов и возврат из подпрограмм с помощью инструкций `call` и `ret`;
- модифицировал программу для вычисления выражения, добавив вложенную подпрограмму и реализовав вычисление композиции функций $f(g(x))$;
- освоил базовые приёмы отладки программ в отладчике GDB: запуск программы, установка и просмотр точек останова, пошаговое выполнение, дизассемблирование кода, переключение синтаксиса команд, использование режима псевдографики;
- научился просматривать и изменять содержимое регистров и ячеек памяти, а также анализировать расположение аргументов командной строки в стеке;
- выполнил задания для самостоятельной работы: преобразовал программу из ЛР8, выделив вычисление функции $f(x)$ в отдельную подпрограмму, и с помощью GDB нашёл и исправил ошибку в программе вычисления выражения $(3 + 2) * 4 + 5$.

В результате работы были укреплены навыки структурирования программ с использованием подпрограмм и навыки отладки на низком уровне, что важно для понимания процессов выполнения программ и эффективного поиска ошибок.