Лабораторная работа №9

Понятие подпрограммы. Отладчик GDB

Хасанов Тимур

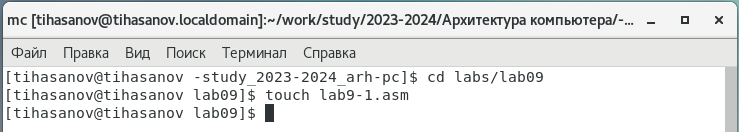
Содержание

# 1 Цель работы

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

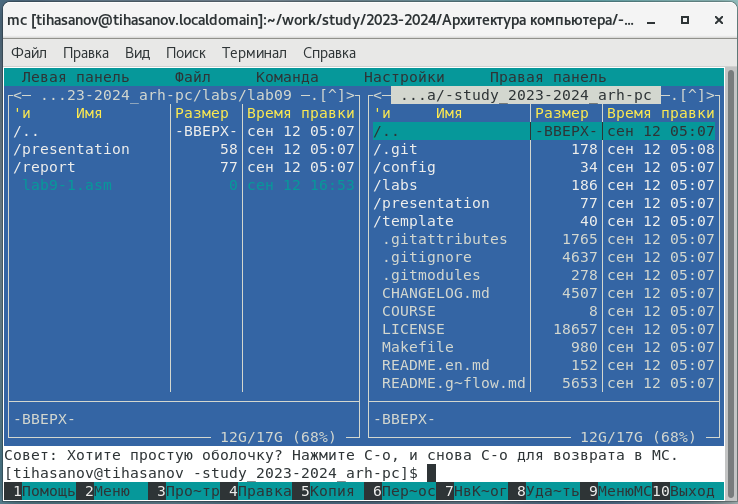
# 2 Выполнение лабораторной работы

Для начала выполнения работы необходимо перейти в рабочую папку и создать файл lab9-1.asm (Рис. 2.1):



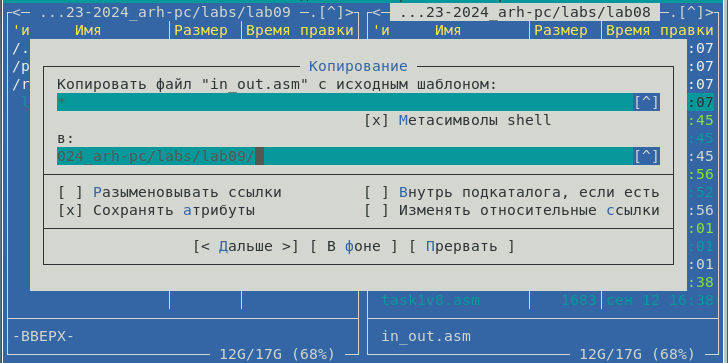
Создание файла lab9-1.asm

Далее, запустим Midnight commander (Рис. 2.2):



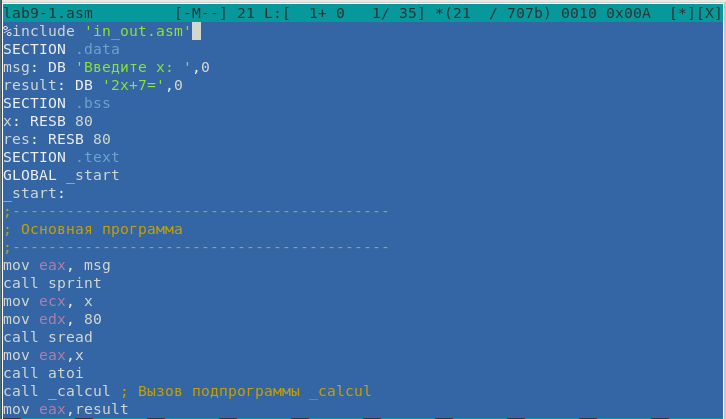
Запуск Midnight commander

Скопируем файл in\_out.asm из директории прошлой работы (Рис. 2.3):



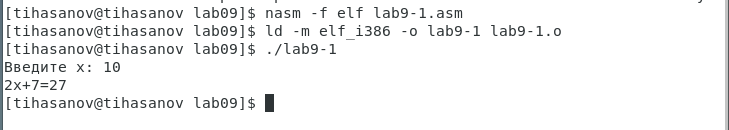
Копирование файла in\_out.asm в рабочую директорию

Вставим в файл lab9-1.asm код из листинга 9.1 (Рис. 2.4):



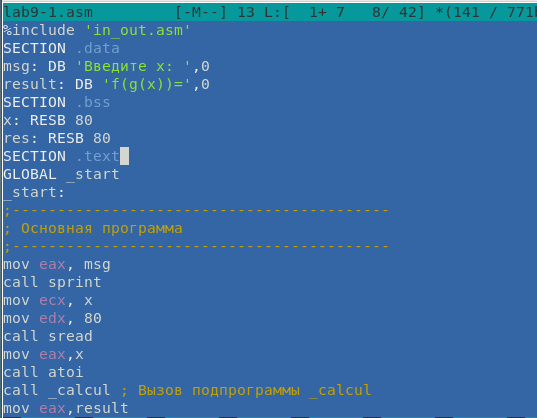
Вставка кода из файла листинга 9.1

Соберём программу и посмотрим на вывод (Рис. 2.5):



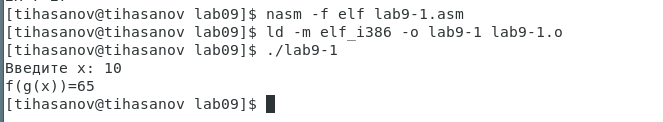
Сборка программы из файла lab9-1.asm и её запуск

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. 2.6):



Изменение файла lab9-1.asm

Соберём программуц и проверим её работу (Рис. 2.7):



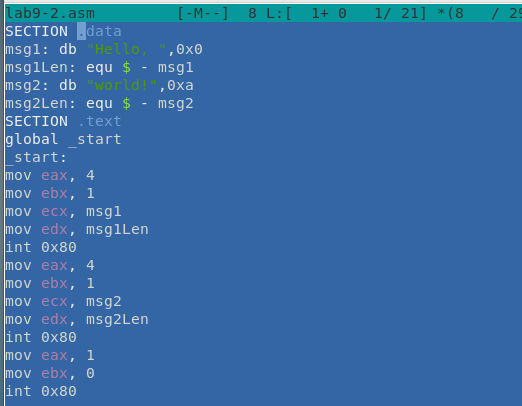
Повторная сборка программы из файла lab9-1.asm и её запуск

Создадим новый файл (Рис. 2.8):

Создание второго файла: lab9-2.asm

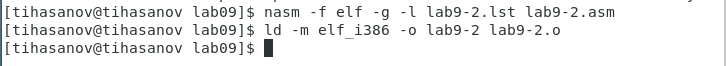
Создание второго файла: lab9-2.asm

Вставим в него код из листинга 9.2 (Рис. 2.9):



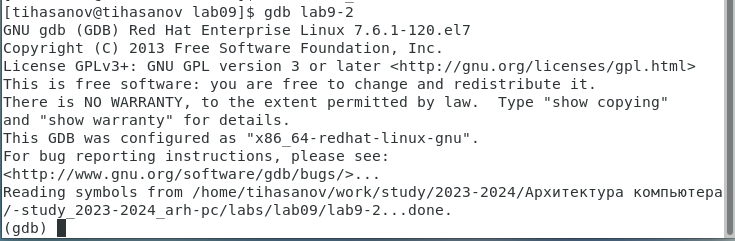
Запись кода из листинга 9.2 в файл lab9-2.asm

Соберём программу следующим образом (с использованием аргумента -g) (Рис. 2.10):



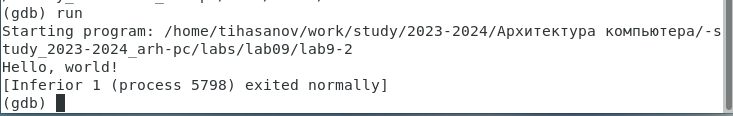
Сборка программы из файла lab9-2.asm

Теперь загрузим её в gdb (Рис. 2.11):



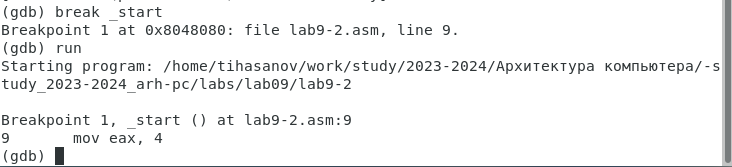
Загрузка программы lab9-2.asm в gdb

Запустим её в отладчике с помощью команды run (Рис. 2.12):



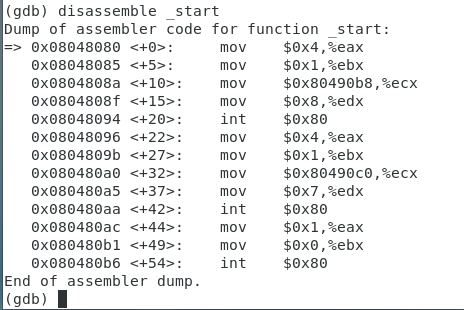
Запуск программы в отладчике

Создадим брейкпоинт на метке \_start с помощью команды break (Рис. 2.13):



Создание брейкпоинта

С помощью команды disassemble дизассемблируем её (Рис. 2.14):



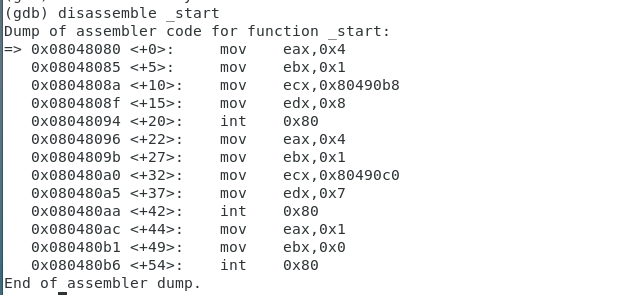
Дизассемблирование программы

Переключим синтаксис вывода на intel (Рис. 2.15):

Переключение на синтаксис intel

Переключение на синтаксис intel

Повторно дизассемблируем программу (Рис. 2.16):



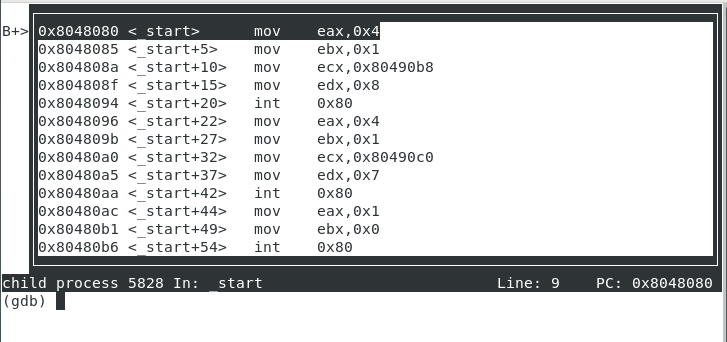
Повторное дизассемблирование

Включим графическое отображения кода (Рис. 2.17):

Включение графического отображения кода и выполнения команд

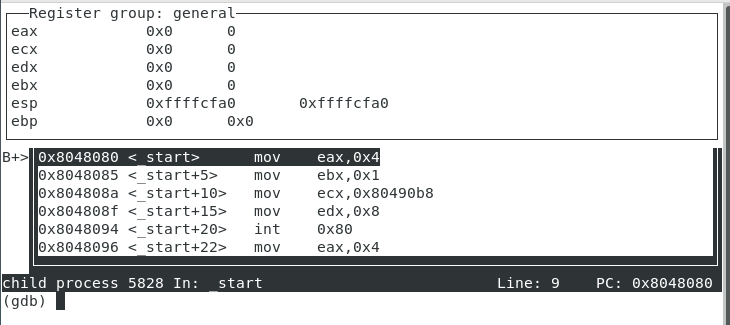
Включение графического отображения кода и выполнения команд

Вот как теперь это выглядит (Рис. 2.18):



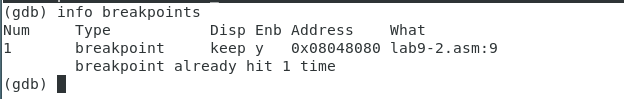
Внешний вид интерфейса

Теперь включеним графическое отображение значений регистров (Рис. 2.19):



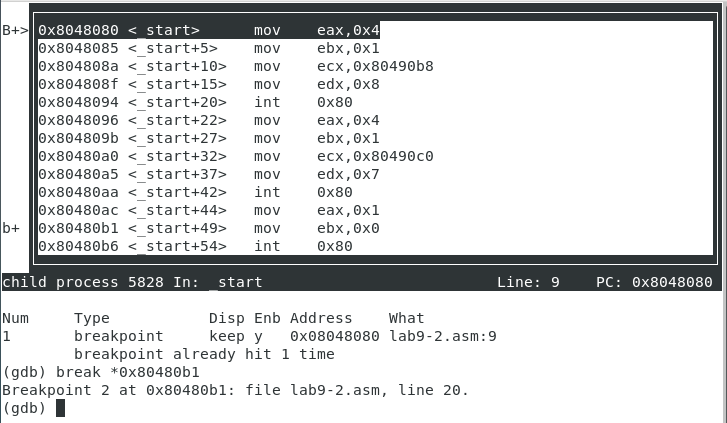
включение графического отображения значений регистров и отображение интерфейса

Выведем инормацию о всех брейкпоинтах (Рис. 2.20):



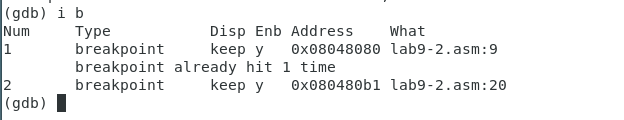
Вывод информации о брейкпоинтах

Попробуем теперь создать брейкпоинт по адресу (Рис. 2.21):



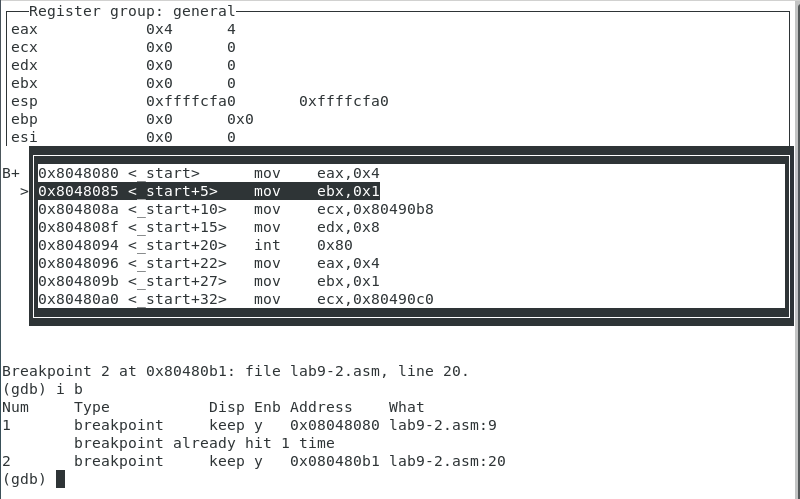
Создание брейкпоинта по адресу

Повторно выведем информацию о брейкпоинтах (Рис. 2.22):

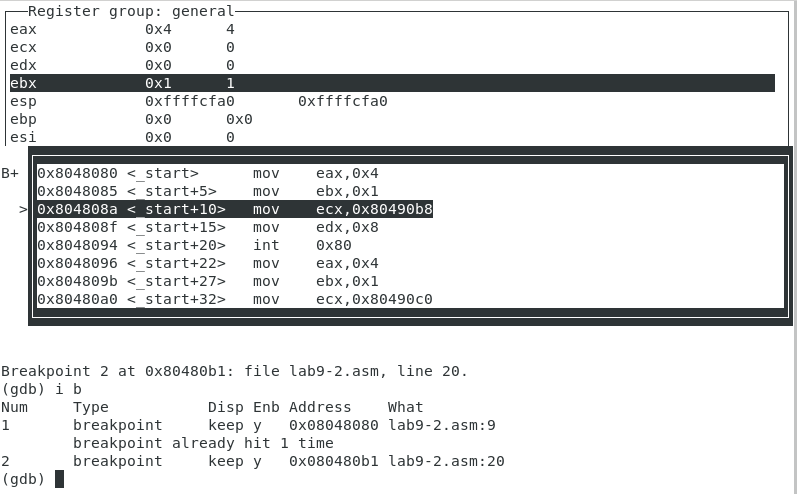


Повторный вывод информации о брейкпоинтах

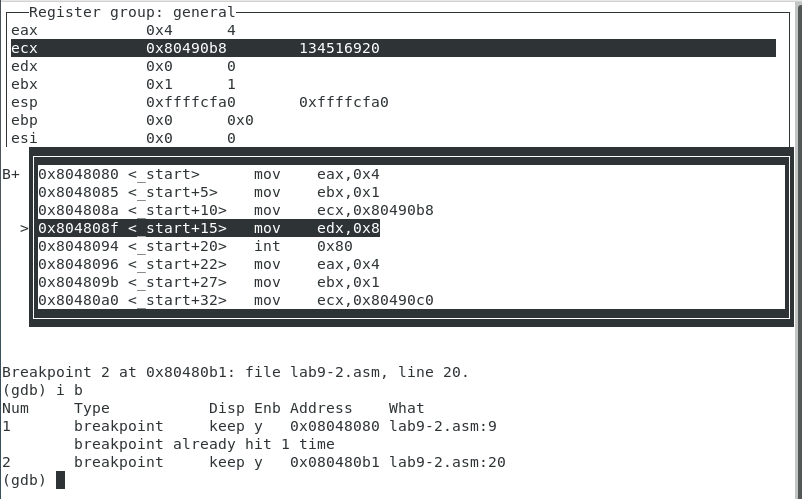
Теперь 5 раз выполним команду si для построчного выполнения кода (Рис. 2.23 - 2.27):



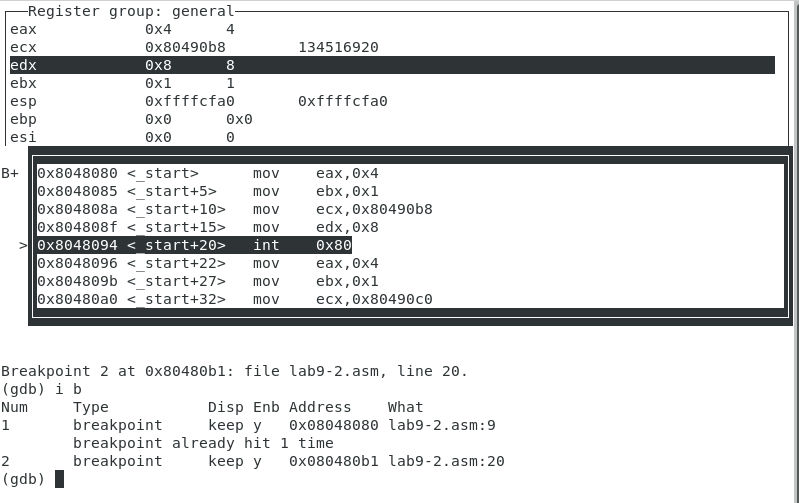
Выполнение следующей команды в коде программы (1)



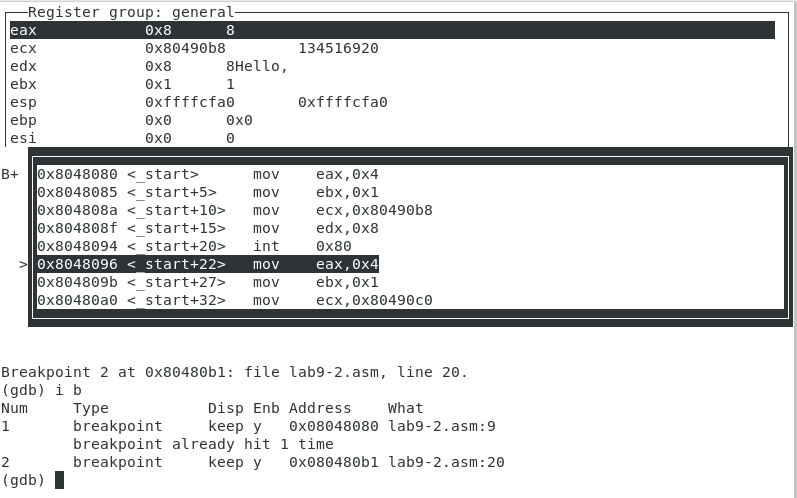
Выполнение следующей команды в коде программы (2)



Выполнение следующей команды в коде программы (3)



Выполнение следующей команды в коде программы (4)



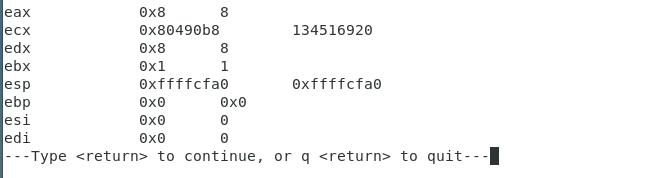
Выполнение следующей команды в коде программы (5)

Как видим, поменялись значения регистров eax, ecx, edx и ebx. Теперь выведем информацию о значениях регистров (Рис. 2.28):

Вывод значений регистров

Вывод значений регистров

Вот, что нам выводится (Рис. 2.29):



Значения регистров

Попробуем вывести значени переменной по имени (Рис. 2.30):

Вывод значения переменной по имени

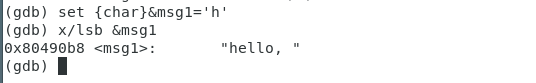
Вывод значения переменной по имени

Теперь попробуем вывести значени переменной по адресу (Рис. 2.31):

Вывод значения переменной по адресу

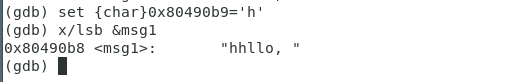
Вывод значения переменной по адресу

Теперь изменим первый символ переменной (Рис. 2.32):



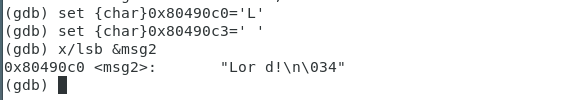
Изменение первого символа переменной по имени и вывод переменной

А теперь изменим второй символ переменной, уже обратясь по адресу (Рис. 2.33):



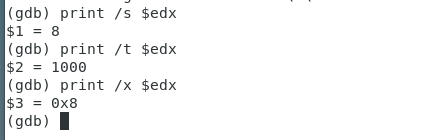
Изменение второго символа переменной по адресу и вывод переменной

Теперь изменим несколько символов второй переменной (Рис. 2.34):



Исменение нескольких символов второй переменной по адресу и вывод переменной

Теперь попробуем вывести значение регистра в строковом, двоичном и шестнадцатиричном виде (Рис. 2.35):



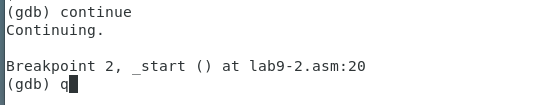
Вывод значения регистра в строковом, двоичном и шестнадцатиричном виде

Попробуем теперь изменить значение регистра (Рис. 2.36):



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 с помощью continue (чтобы продолжить выполнение) и выйдем из отладчика (Рис. 2.37 - 2.38):

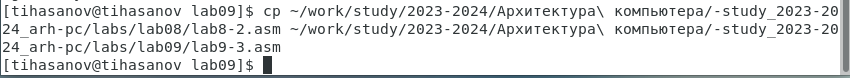


Завершение работы программы

Завершение работы программы (продолжение)

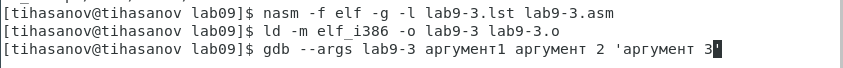
Завершение работы программы (продолжение)

Скопируем файл из прошлой работы (Рис. 2.39):



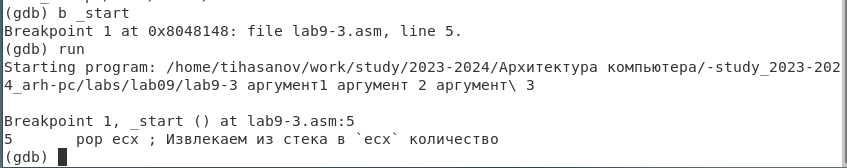
Копирование файла из прошлой работы

Соберём его и вгрузим в gdb (Рис. 2.40):



Сборка программы и вгрузка в gdb

Создадим брейкпоинт и запустим программу (Рис. 2.41):



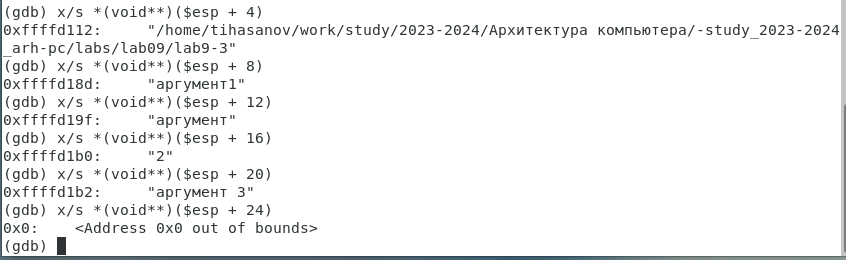
Создание брейкпоинта и запуск программы

Теперь выведем значение регистра esp, где хранятся данные о стеке (Рис. 2.42):

Вывод значения регистра esp

Вывод значения регистра esp

Теперь выведем значение всех элементов стека (Рис. 2.43):



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

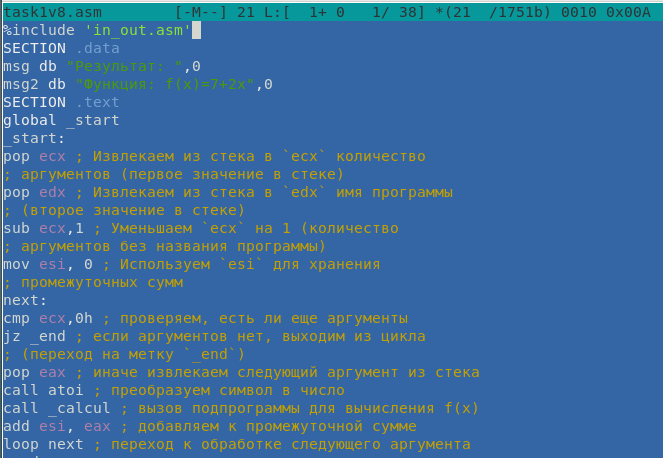
# 3 Выполнение задания для самостоятельной работы

Скопируем файл первого задания прошлой самостоятельной работы (Рис. 3.1):

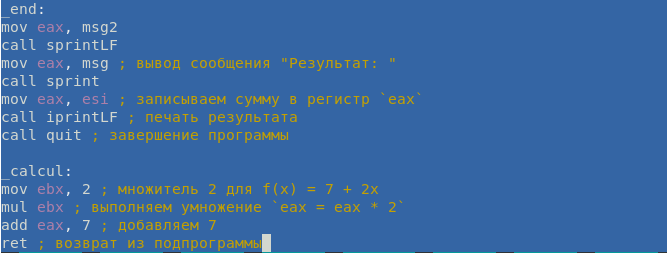
|  |
| --- |
| Копирование первого файла самостоятельной работы из прошлой работы |

Копирование первого файла самостоятельной работы из прошлой работы

Нам нужно переписать его так, чтобы он использовал для авчисления выражения подпрограмму (Рис. 3.2 - 3.3):

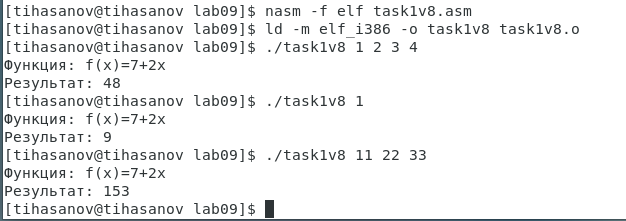


Редактирование кода



Редактирование кода (продолжение)

Соберём его и проверим корректность выполнения (Рис. 3.4):



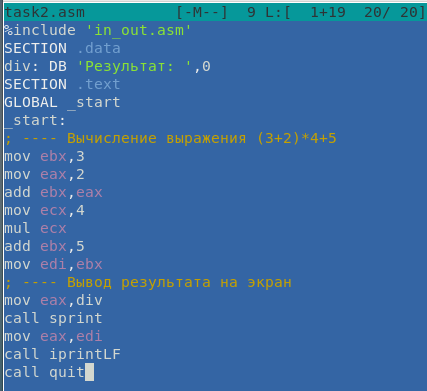
Сборка и проверка работы программы

Создадим файл второго задания самостоятельной работы (Рис. 3.5):

Создание файла второго задания самостоятельной работы

Создание файла второго задания самостоятельной работы

Вставим в него код из листинга 9.3 (Рис. 3.6):



Вставка кода из листинга 9.3

Соберём его (Рис. 3.7):

Сборка программы

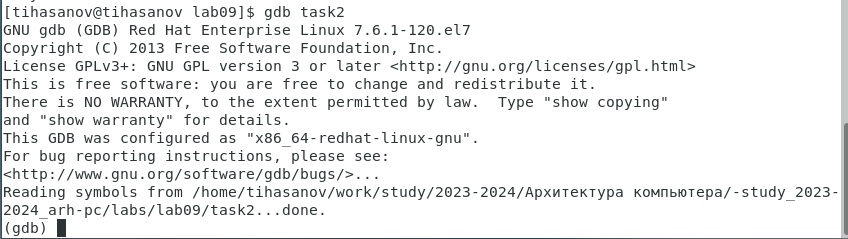
Сборка программы

И запустим (Рис. 3.8):

Запуск программы

Запуск программы

Как видим, код считает значение выражения неправильно. Загрузим его в gdb (Рис. 3.9):



Вгрузка программы в gdb

Переключим его на синтаксис intel (Рис. 3.10):

Переключение на синтаксис intel

Переключение на синтаксис intel

Включим графическое отображение кода (Рис. 3.11):

Включение графического отображения кода и выполнения команд

Включение графического отображения кода и выполнения команд

Включеним графическое отображение значений регистров (Рис. 3.12):

Включение графического отображения значений регистров

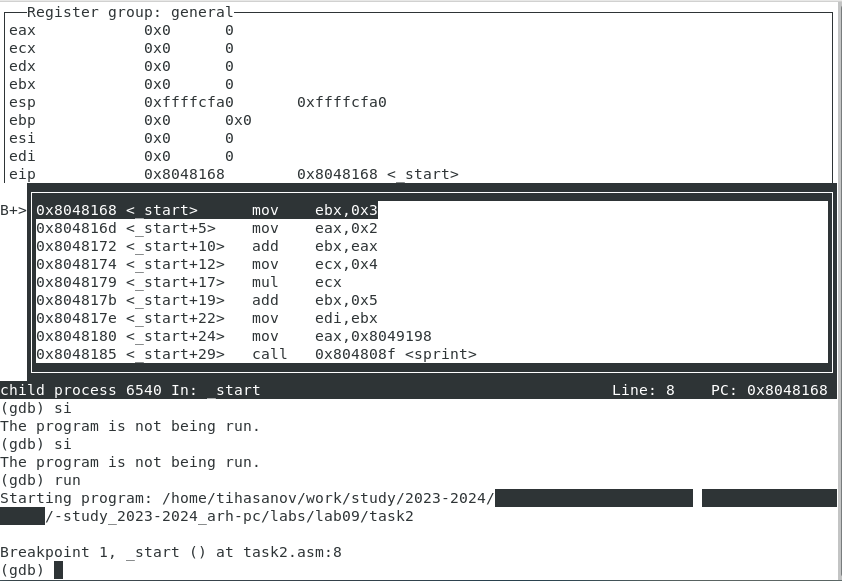
Включение графического отображения значений регистров

Установим брейкпоинт на \_start (Рис. 3.13):

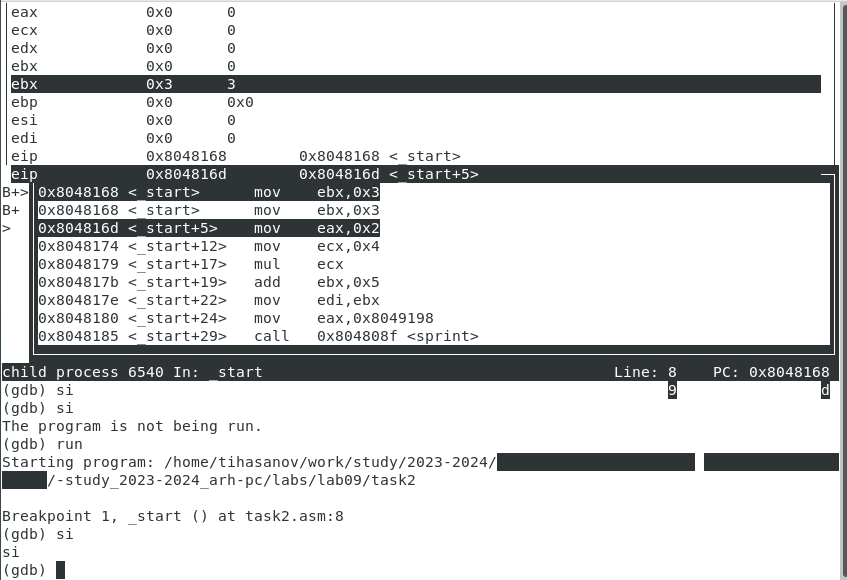
Установка брейкпоинта

Установка брейкпоинта

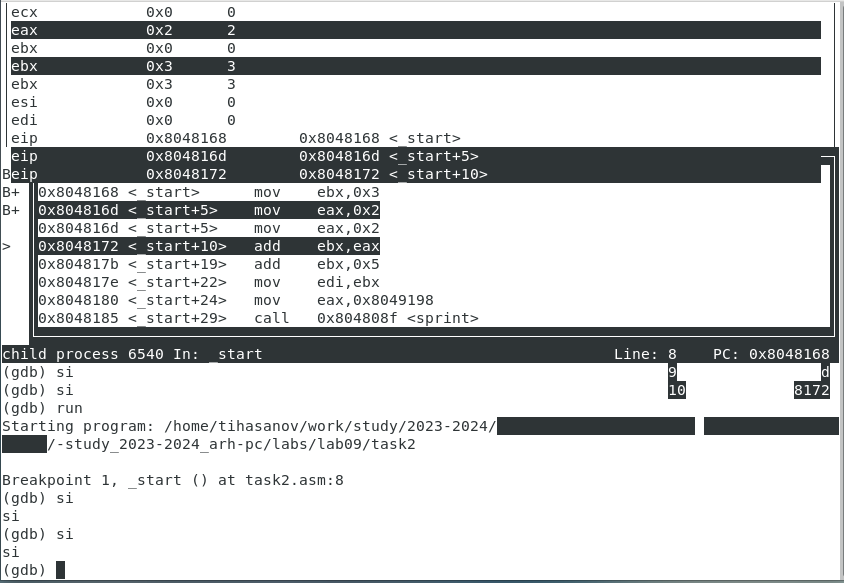
И начнём построчно выполнять код (Рис. 3.14 - 3.20):



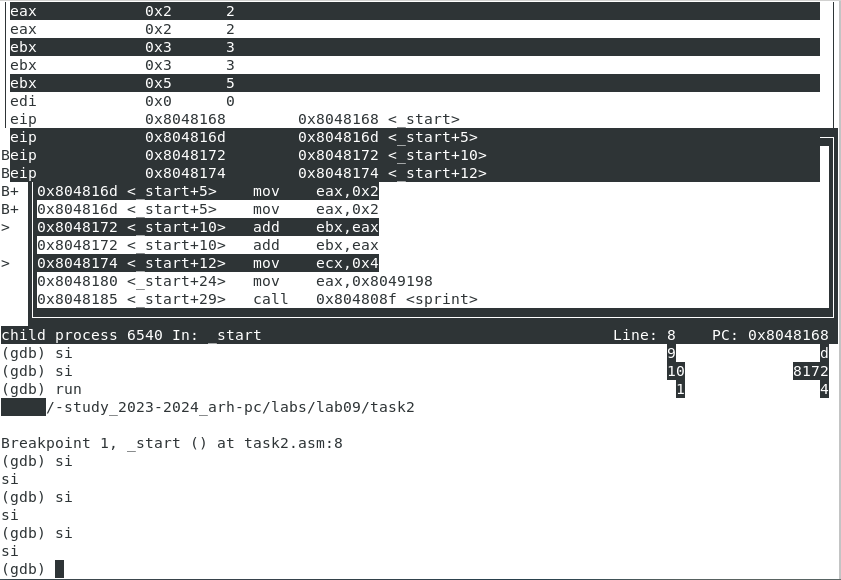
Значение всех регистров на 0 шаге



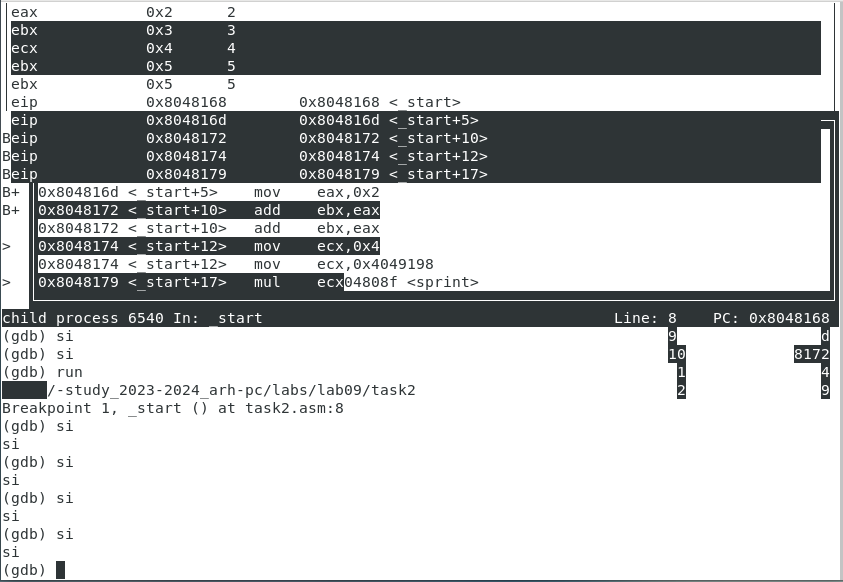
Значение всех регистров на 1 шаге



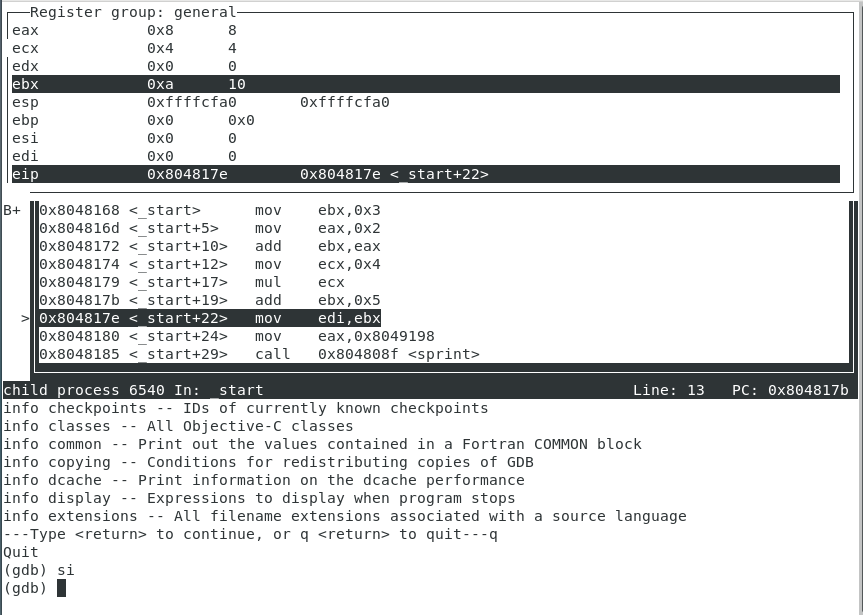
Значение всех регистров на 2 шаге



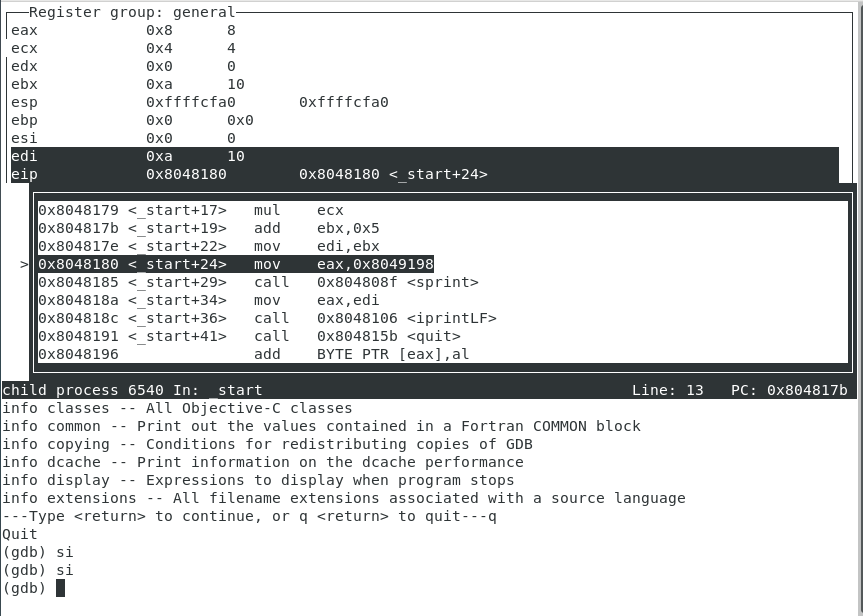
Значение всех регистров на 3 шаге



Значение всех регистров на 4 шаге

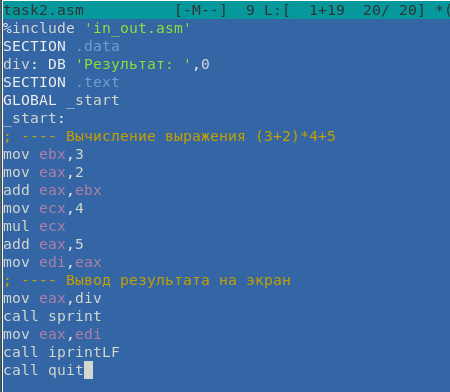


Значение всех регистров на 5 шаге



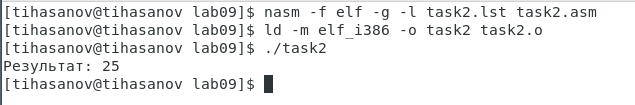
Значение всех регистров на 6 шаге

Как видим, мы должны были умножить значение регистра ebx, но умножили регистр eax. Нам необходимо все результаты хранить в регистре eax. Изменим код (Рис. 3.21):



Редактирование кода

И проверим корректность его выполнения (Рис. 3.22):



Сборка кода и проверка выполнения

Как видим, теперь код работает корректно

# 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