Отчёт по лабораторной работе №9

Понятие подпрограммы. Отладчик GDB.

Малкина Дарья Александровна

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

1. Сначала создадим каталог lab09 и файл lab9-1.asm. Рассмотрим программу, вычисляющую f(x)=2x+7 с помощью подпрограммы \_calcul. Добавим в файл lab9-1.asm текст программы из листинга 9.1. Создадим исполняемый файл и проверим его работу:

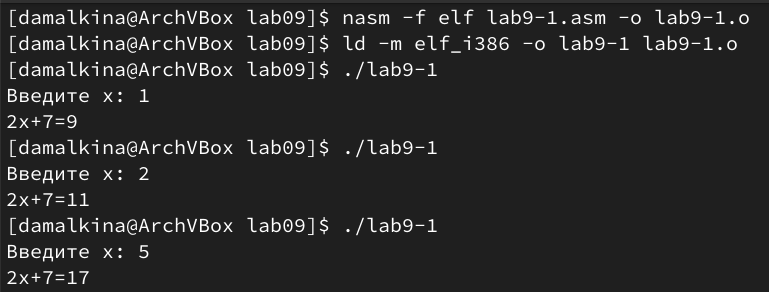


Рис. 1: Программа lab9-1 - f(x)=2x+7

После этого добавим в программу подпрограмму \_subcalcul внутри \_calcul, чтобы \_subcalcul вычисляла g(x) = 3x-1, а после передавала значение x в \_calcul, где будет вычислено f(g(x)):

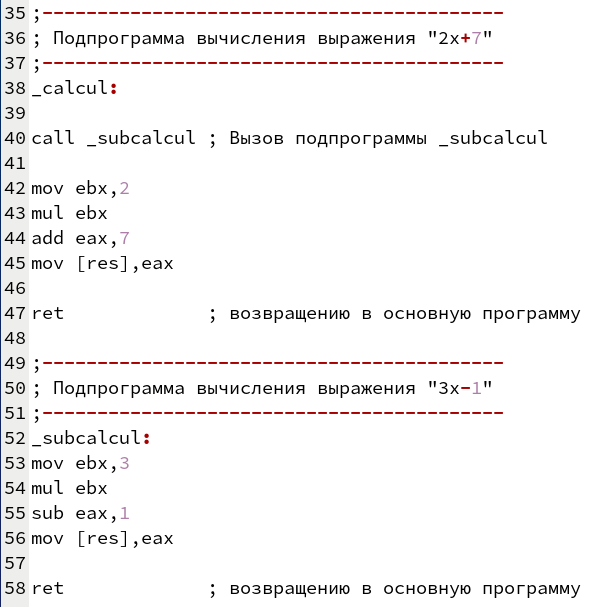


Рис. 2: Добавление подпрограммы \_subcalcul

Создадим исполняемый файл и проверим его работу:

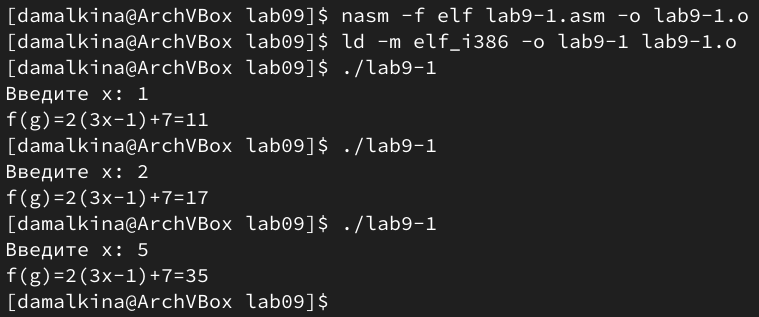


Рис. 3: Программа lab9-1 - f((g)x)

## 2.1 Отладка программам с помощью GDB

1. Создадим файл lab9-2.asm с программой из листинга 9.2 и соберем исполняемый файл. Добавим отладочную информацию, используя ключ -g:

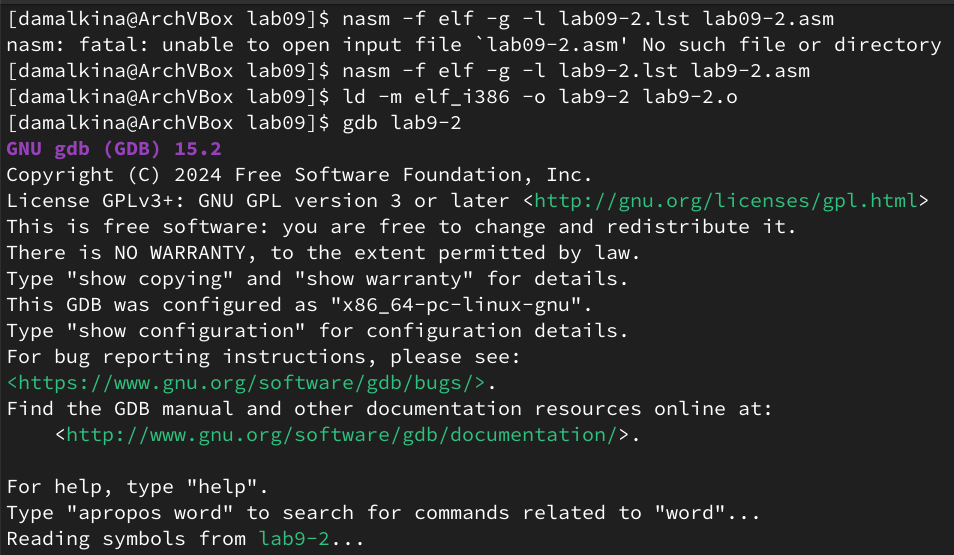


Рис. 4: Отладка программы lab9-2 с помощью GDB

После этого загрузим исполняемый файл в отладчик GDB и проверим работу программы командой run:

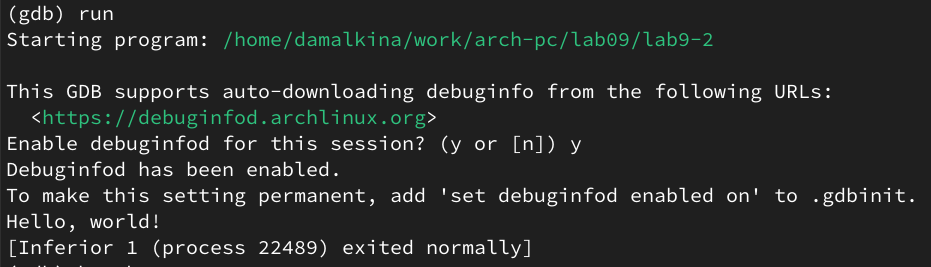


Рис. 5: Запуск программы lab9-2 в оболчке GDB

Далее установим точку останова на метке \_start командой break \_start и запустим программу еще раз:



Рис. 6: Установка точки останова на метке \_start

Затем посмотрим дизассемблированный код командой disassemble \_start:



Рис. 7: Дизассемблированный код программы lab9-2

Переключимся на синтаксис Intel командой set disassembly-flavor intel и снова выведем дизассемблированный код:



Рис. 8: Дизассемблированный код - синтаксис Intel

Заметим, что отображения кода в режимах ATT и Intel различаются:

Режим ATT - операнды пишутся в порядке источник, затем приемник, при этом перед именем регистра ставится символ %, а непосредственные значения обозначаются знаком $ перед числом.

Режим Intel - операнды пишутся в порядке приемник, затем источник. Префикса % перед регистрами нет и нак $ не используется для обозначения непосредственных значений.

Включим режим псевдографики командами layout asm и layout regs:

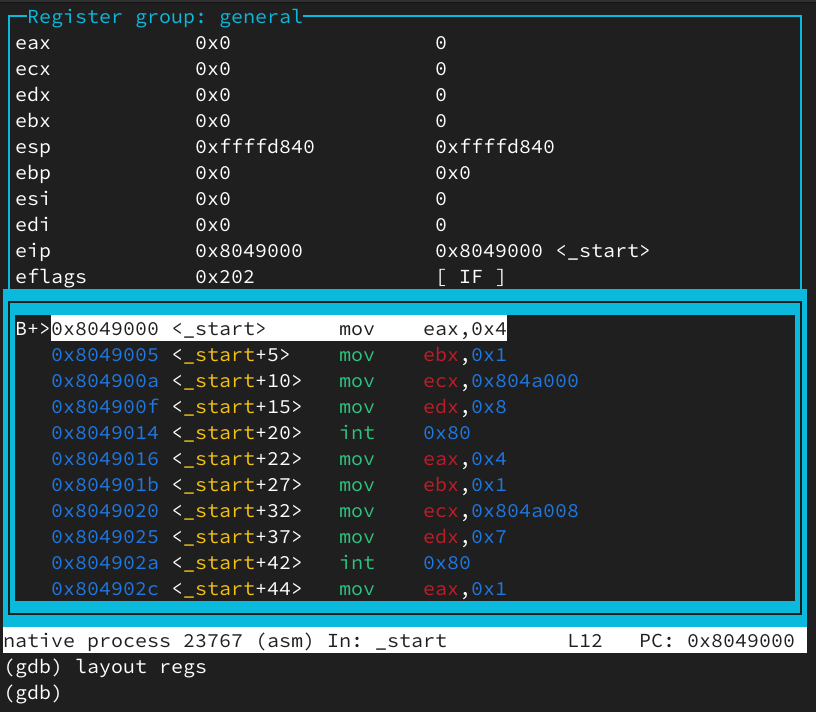


Рис. 9: Режим псевдографики

### 2.1.1 Добавление точек останова

До этого мы установили точку останова на метке \_start, проверим это командой info breakpoints (i b). Затем по адресу инструкции установим еще одну точку останова командой break \*. После посмотрим информацию о всех установленных точках останова командой i b:

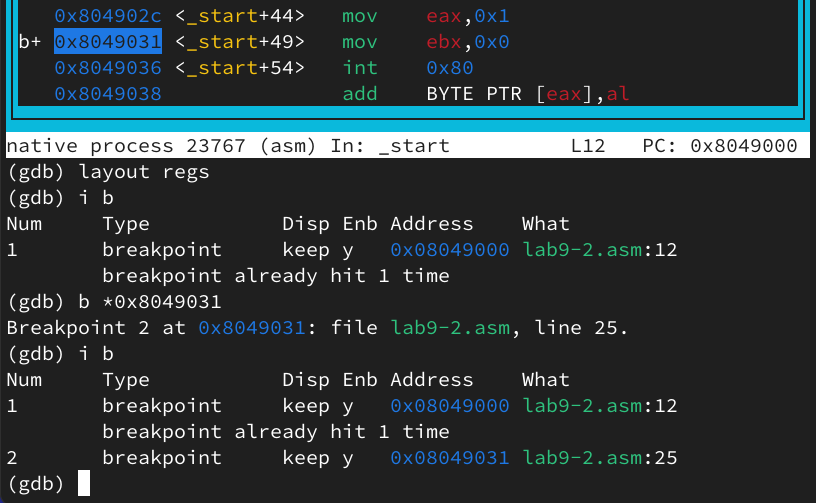


Рис. 10: Точки останова

### 2.1.2 Работа с данными программы в GDB

Воспользуемся командой stepi (si) пять раз, и понаблюдаем, как меняются значения регистров после каждой инструкции:

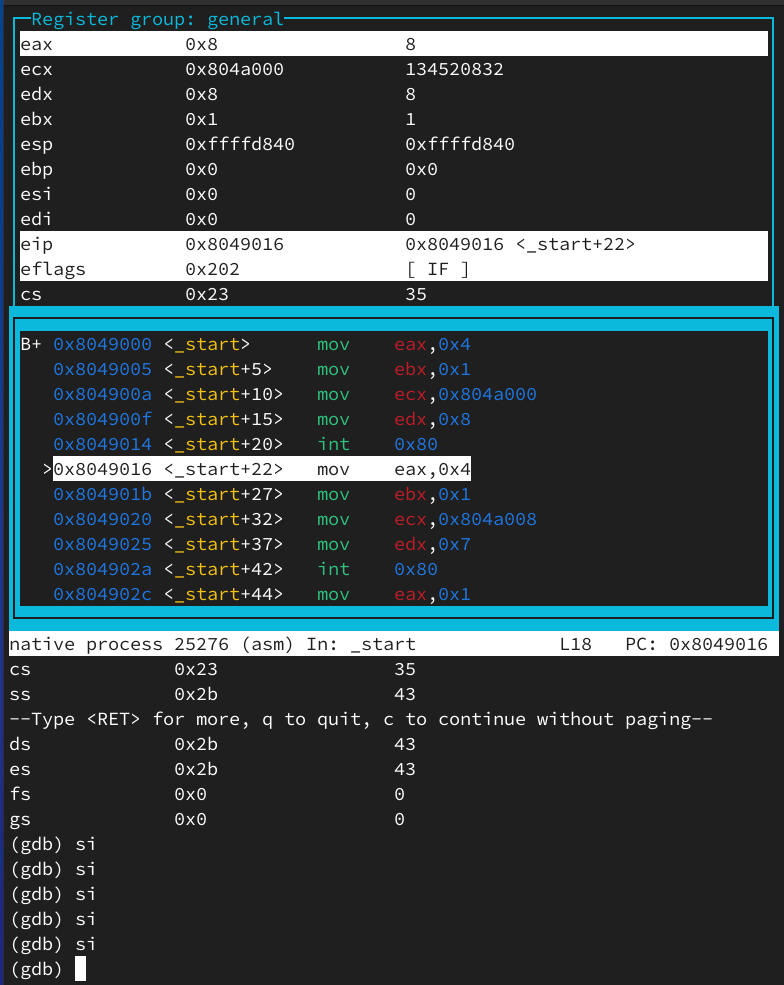


Рис. 11: Команда stepi

Воспользуемся командой x/1sb &msg1 для просмотра значения msg1. Затем мы найдем значение msg2 по адресу (определим адресс в дизассемблированной инструкции).

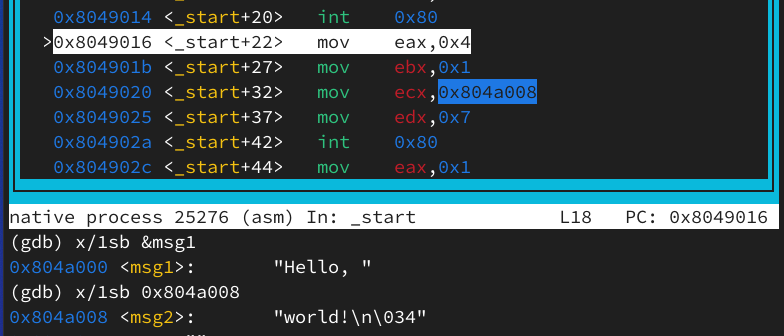


Рис. 12: Просмотр значений msg1 и msg2

Изменим первый символ msg1 командой set {char}msg1=‘h’, после аналогично заменим первый символ в переменной msg2:

|  |
| --- |
| Рис. 13: Изменение переменных - команда set |

Рис. 13: Изменение переменных - команда set

Выведем в различных форматах (в шестнадцатеричном формате, в двоичном формате и в символьном виде) значение регистра edx:

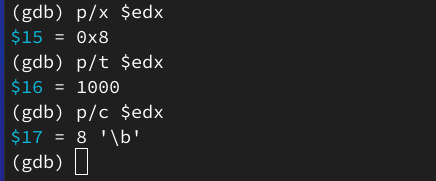


Рис. 14: Команда print

С помощью команды set изменим значение регистра ebx сначала на ‘2’,затем на 2, и после каждого изменения выведем значение edx как строку:

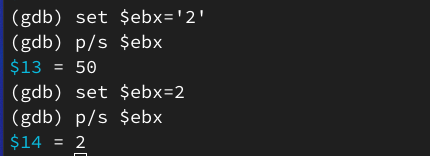


Рис. 15: Команда print

Заметим разницу в выводе:

В первом случае set $ebx=‘2’ - мы присвоили регистру ebx строковое значение ‘2’, поэтому команда p/s $ebx, которая выводит значение как строку, отображает ASCII-код символа “2”, который равен 50.

Во втором случае set $ebx=2 - мы присвоили регистру ebx числовое значение 2, теперь команда p/s $ebx выводит число 2, потому что интерпретирует ebx как число, а не как строку.

Выходим из GDB.

### 2.1.3 Обработка аргументов командной строки в GDB

Скопируем файл lab8-2.asm в файл lab09-3.asm, создадим исполняемый файл:

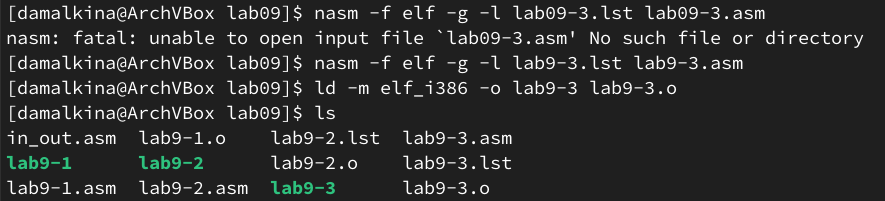


Рис. 16: Создаём файл lab9-3.as

После загрузим исполняемый файл в gdb с помощью ключа –args, передав ему аргументы:

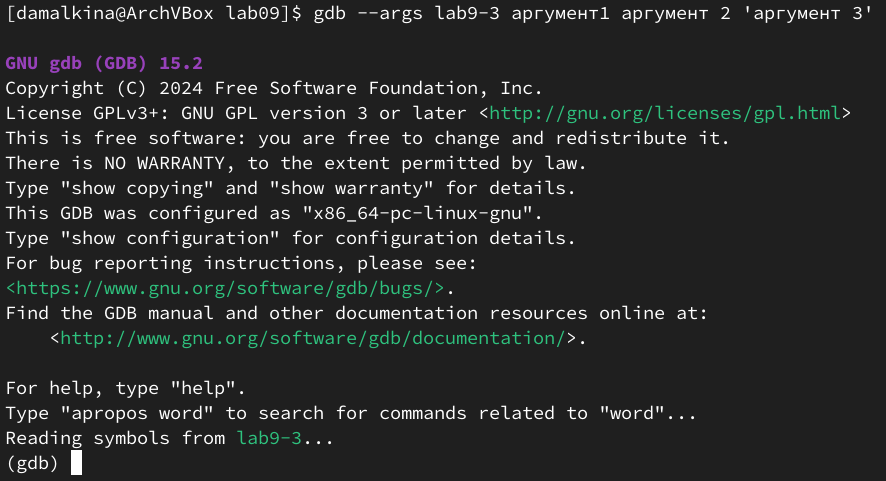


Рис. 17: Загружаем файл в GDB

Установим точку останова перед \_start и запустим программу:

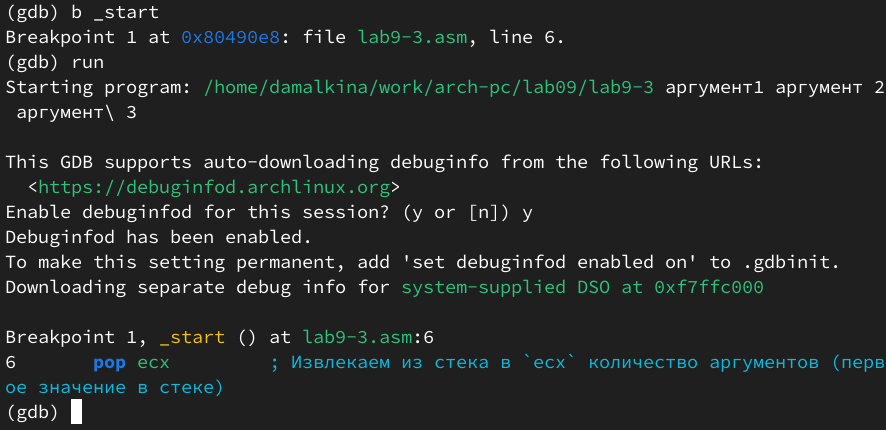


Рис. 18: Точка останова перед \_start

Проверим количество аргументов с помощью команды x/x $esp. Результат - число 5, так как у нас есть имя программы и четыре аргумента. Далее, исследуем содержимое стека. По адресу [esp+4] находится адрес имени программы, по адресу [esp+8] — адрес первого аргумента, по адресу [esp+12] — второго, и так далее:

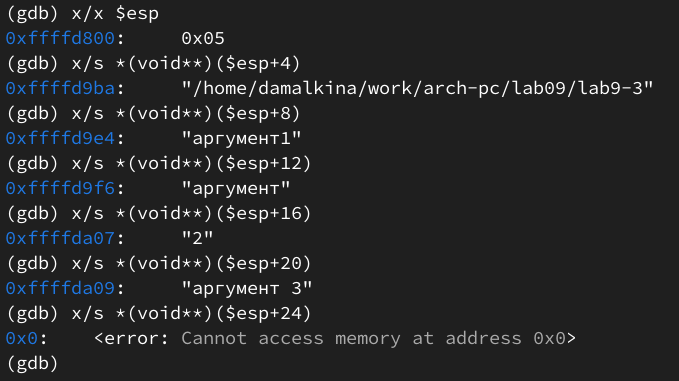


Рис. 19: Содержимое стека

Так как в 32-битных системах, адреса памяти в виде 32-битными числами(32 бита = 4 байта), поэтому мы перемещаемся по стеку с интервалом в 4 байта вперед, чтобы добраться до начала следующего 32-битного адреса.

# 3 Выполнение задания для самостоятельной работы

1. Преобразуем программу из лабораторной работы №8 (Задание №1 для самостоятель- ной работы), реализовав вычисление значения функции f(x) как подпрограмму.

В сегменте .bss объявим переменную res для хранения результатов вычислений:

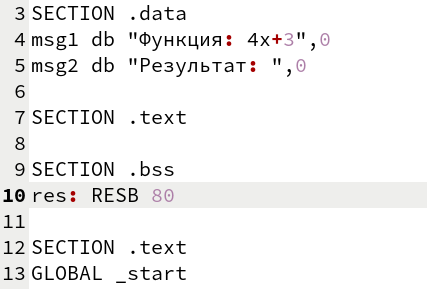


Рис. 20: Переменная res

В основной программе добавим вызов подрограммы \_funcalcul (33 строка) и изменим операнды в сложении промежуточных результатов функции (35 строка):

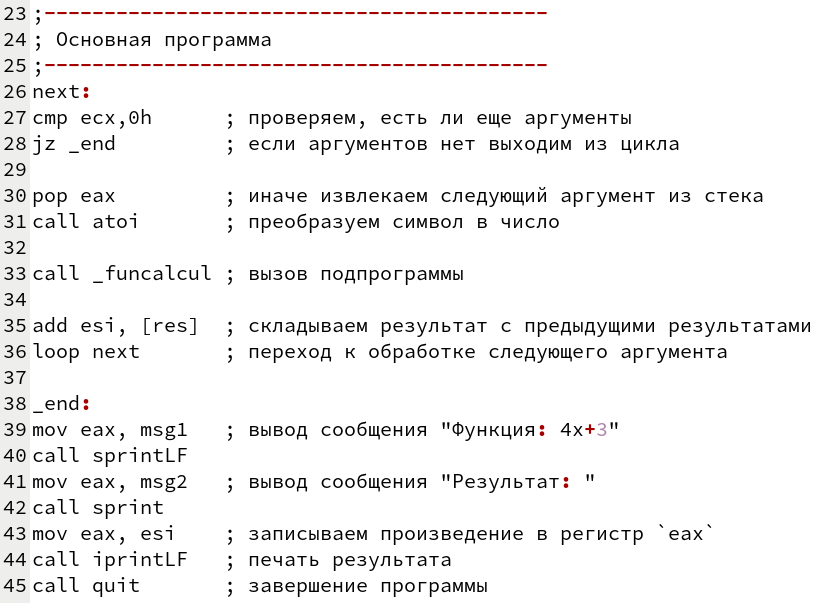


Рис. 21: Основная программа

Вынесем вычисления функции за основной код, добавим сохранение результата в переменную res (54 строка):

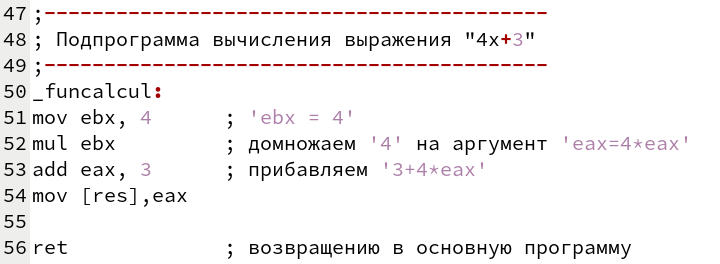


Рис. 22: Подпрограмма \_funcalcul

Создаём исполнительный файл и запускаем его, проверим работу программы с разными аргументами, для простоты проверки введем те же аргументы, что и при проверке программы lab8-var5:

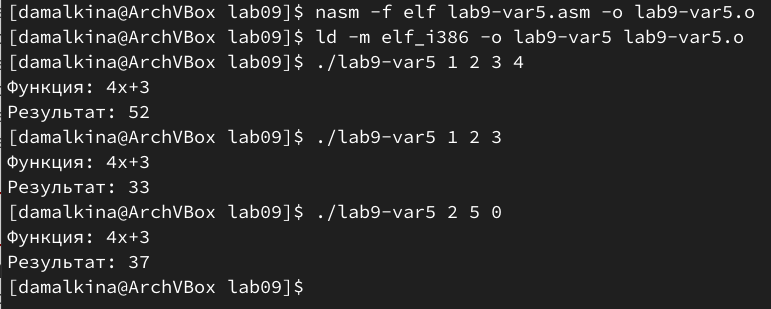


Рис. 23: Программа lab9-var5

Для сравнения приведем результаты работы программы lab8-var5:

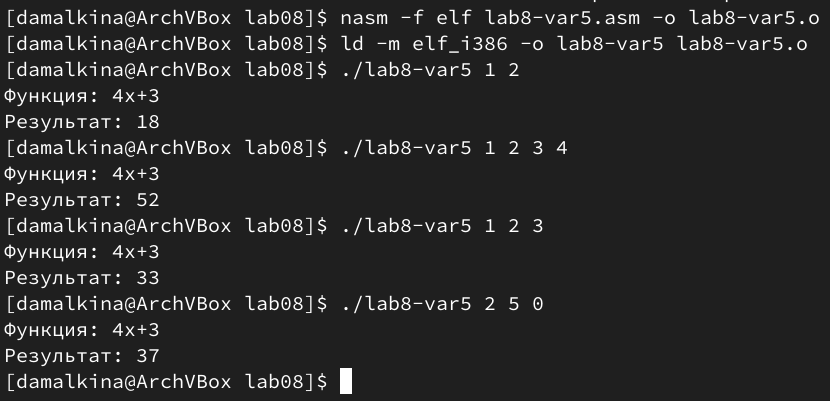


Рис. 24: Программа lab8-var5

1. В листинге 9.3 приведена программа вычисления выражения (3+2)∗4+5. При запуске данная программа дает результат 10, что неверно, убедимся в этом, посчитав аналитически 5\*4+5=20+5=25:

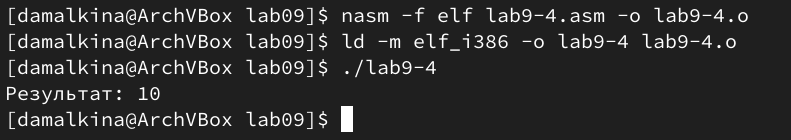


Рис. 25: Неверный результат

С помощью отладчика GDB, проанализируем изменения значений регистров и определим ошибку.

Соберем исполняемый файл, добавив отладочную информацию, после загрузим исполняемый файл в отладчик GDB и проверим работу программы:

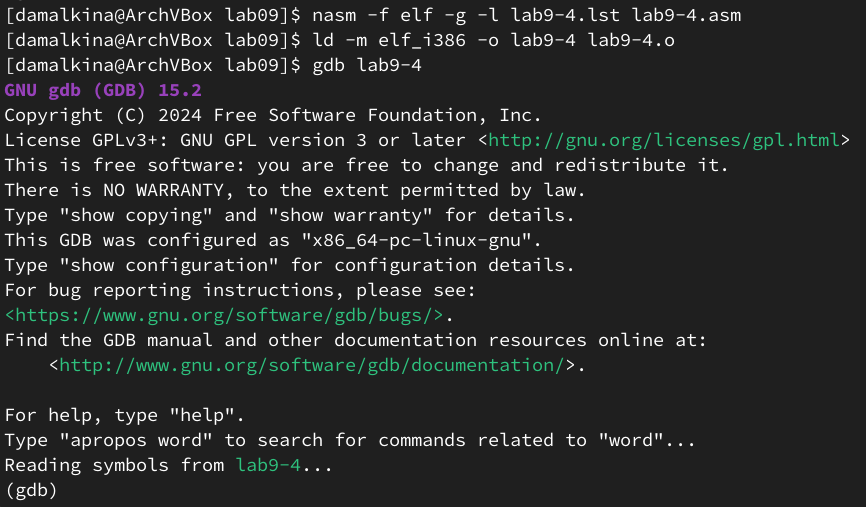


Рис. 26: Запуск программы lab9-2 в оболчке GDB

Установим точку останова на метке \_start и запустим программу еще раз:

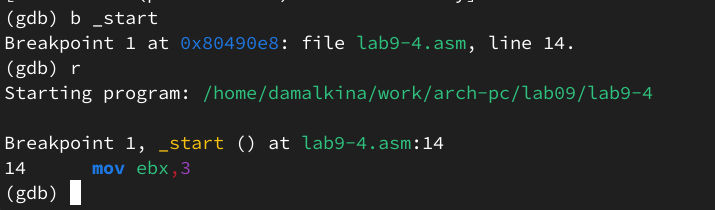


Рис. 27: Установка точки останова на метке \_start

Затем посмотрим дизассемблированный код:



Рис. 28: Дизассемблированный код программы lab9-4

Включим режим псевдографики и добавим точки установа по адресу инструкции после арифметических команд:

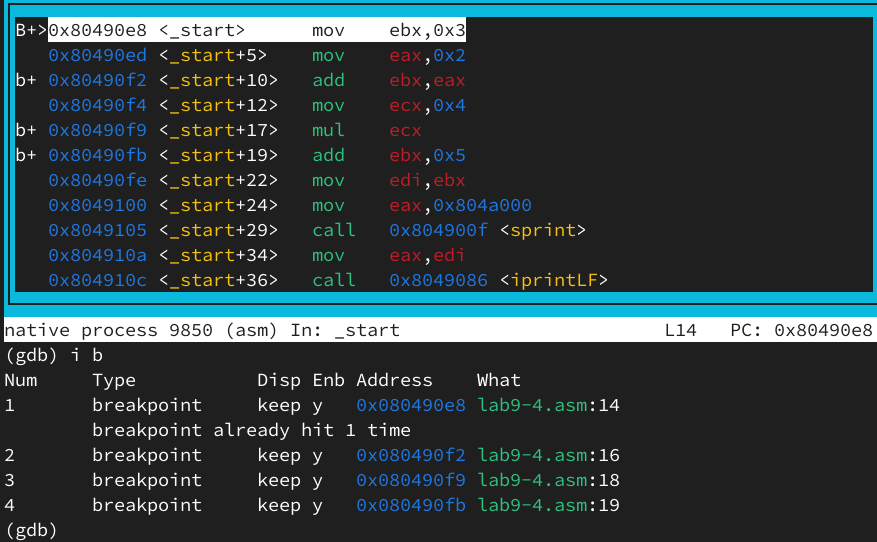


Рис. 29: Добавление точек останова

Воспользуемся командами stepi и print, и понаблюдаем, как меняются значения регистров после инструкций. После инструкции add ebx,eax, регистры принимают верные значения:

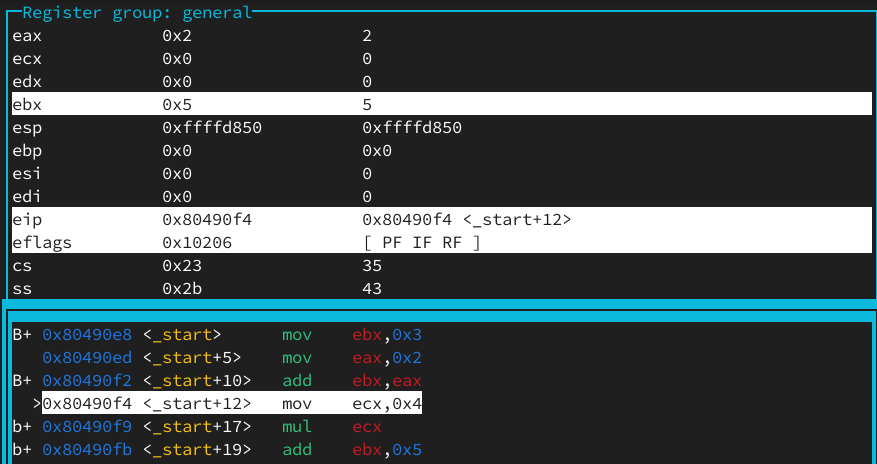


Рис. 30: Значения регистров после add ebx,eax

Заметим, что после команды mul ecx, значение ebx остаётся неизменным, хотя должно принимать значение 5\*4=20, но меняется значение eax, хотя должно оставаться равным 2. Делаем вывод, что инструкция mul ecx умножает содержимое регистра eax на ecx, а не ebx на ecx, как задумано, результат записывается в eax:

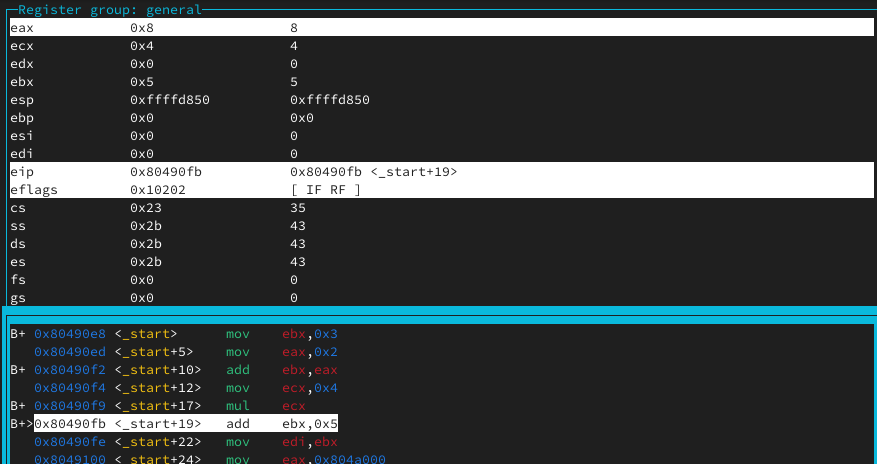


Рис. 31: Первая ошибка - неправильное использование инструкции mul

С помощью команды print сравним значения регистров eax, ebx, ecx до команды mul ecx и после:

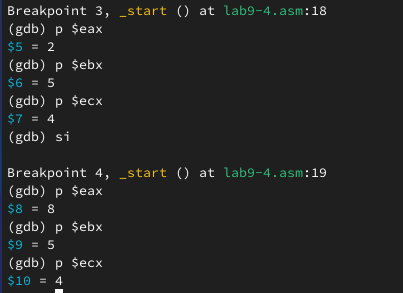


Рис. 32: Значения регистров

Чтобы исправить это, нужно сначала скопировать содержимое ebx в eax перед выполнением mul ecx. Однако код всё равно неверный из-за того, что после умножения 5 прибавляется к ebx, а не к результату умножения, который находится в eax. Это вторая ошибка - неправильное место для второго сложения:

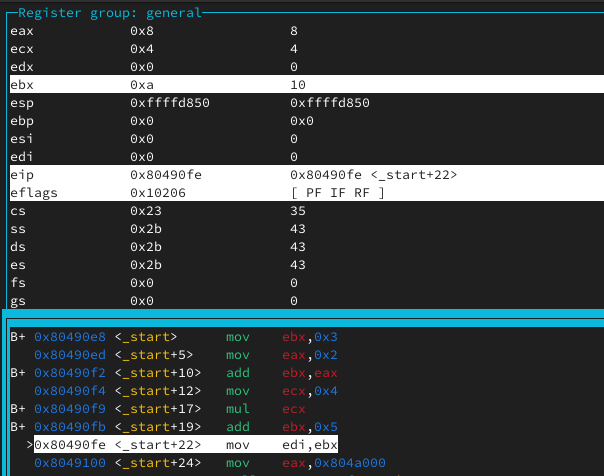


Рис. 33: Значения регистров после add edi,ebx

Исправим код, скопируем значение ebx в eax перед выполнением mul ecx и прибавим 5 напрямую к eax, после чего сохраним результат в edi:

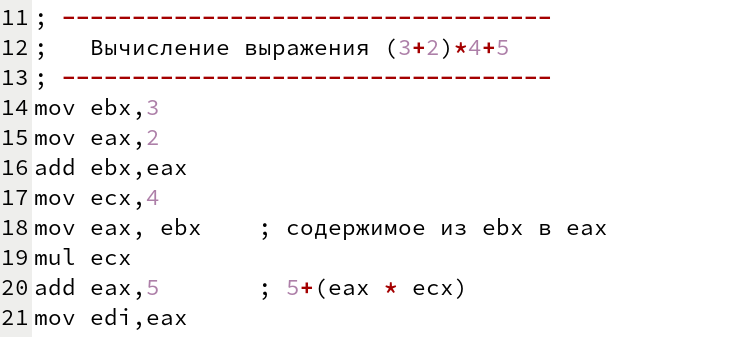


Рис. 34: Исправленный код программы lab9-4

Создим исполнительный файл и запустим его, для проверки работы программы, убедимся, что после внесения изменений программа работает корректно:

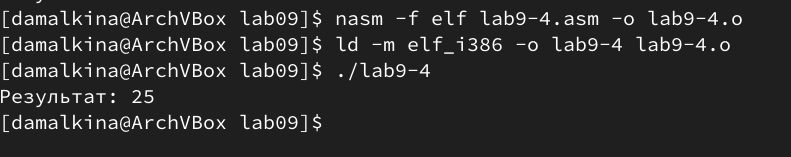


Рис. 35: Корекктная работа программы lab9-4

# 4 Выводы

В ходе лабораторной работы и выполнения самостоятельного задания мы приобрели практические навыки написания ассемблерных программ с использованием подпрограмм, попрактиковались в отладке программ с помощью GDB, научились использовать GDB для пошагового выполнения кода, анализа значений регистров и памяти, что позволило нам обнаружить и исправить ошибки в программе.

# Список литературы