**РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ**

**Факультет физико-математических и естественных наук Кафедра прикладной информатики и теории вероятностей**

**ОТЧЕТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ № 9.**

дисциплина: *Архитектура компьютеров*

Студент: Подхалюзина Виолетта Михайловна

Группа: НКАбд-04-24

**МОСКВА**

2024 г.

Оглавление

[1 Цель работы 2](#_Toc1551896633)

[2 Введение 3](#_Toc859028035)

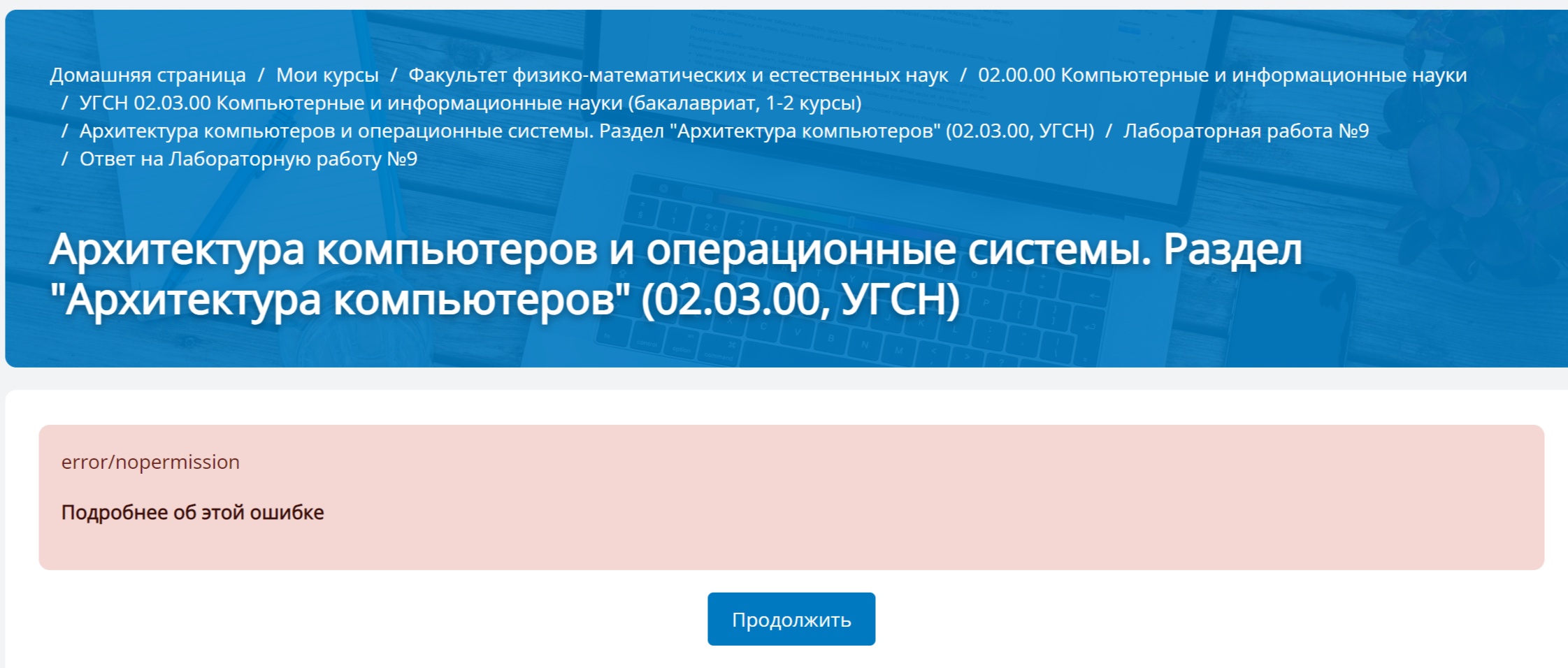
[3 Выполнение лабораторной работы 3](#_Toc345695478)

[3.1 Начало работы 4](#_Toc1864924682)

[3.2 Самостоятельная работа 9](#_Toc1957175182)

[4 Контрольные вопросы для самопроверки 10](#_Toc1558872490)

[5 Список литературы 11](#_Toc63600490)



Прошу обратить ваше внимание. Сайт не давал мне возможности загрузить задание до окончания дедлайна. Я не знаю, с чем это связано, но надеюсь на ваше понимание.

# Цель работы

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

# Введение

Отладка — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

• обнаружение ошибки;

• поиск её местонахождения;

• определение причины ошибки;

• исправление ошибки.

Можно выделить следующие типы ошибок:

• синтаксические ошибки — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка;

• семантические ошибки — являются логическими и приводят к тому, что программа запускается, отрабатывает, но не даёт желаемого результата;

• ошибки в процессе выполнения — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль).

Второй этап — поиск местонахождения ошибки. Некоторые ошибки обнаружить довольно трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга.

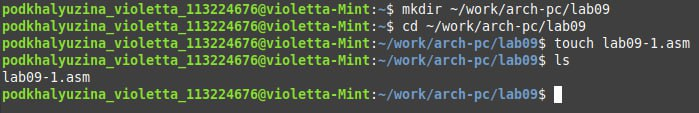
Третий этап — выяснение причины ошибки. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы.

Последний этап — исправление ошибки. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново.

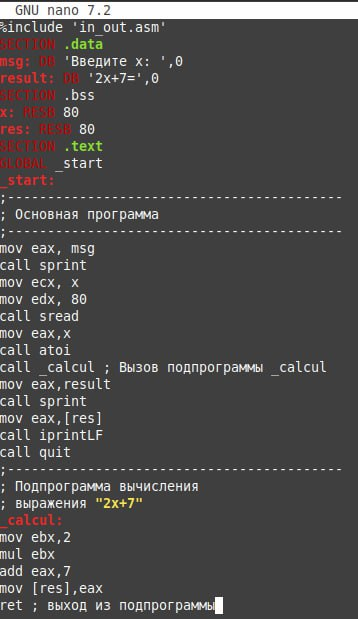
# Выполнение лабораторной работы

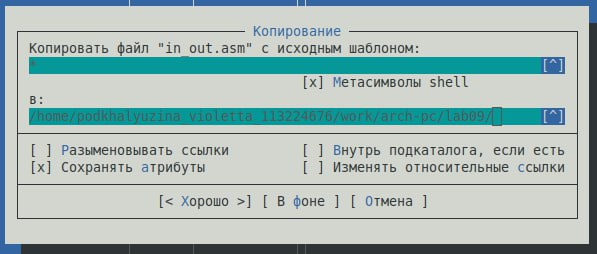
# Начало работы

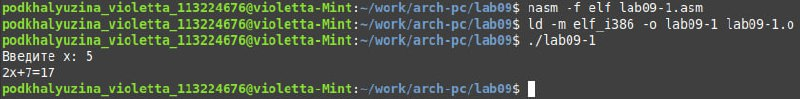
Я создаю каталог для программ лабораторной работы №9, перехожу в него и создаю файл lab9-1.asm

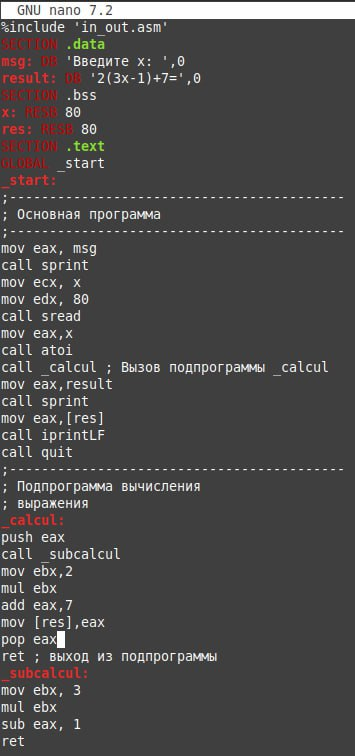


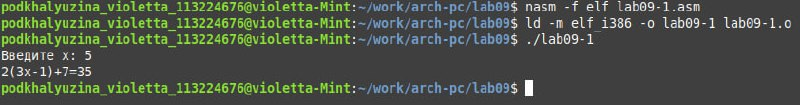
Далее я ввела в файл lab9-1.asm текст программы из листинга 9.1. Создала исполняемый файл и проверила его работу

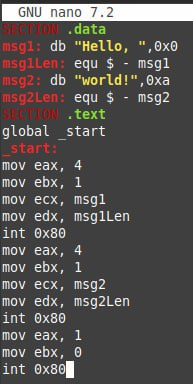
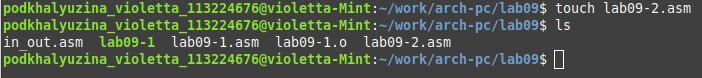




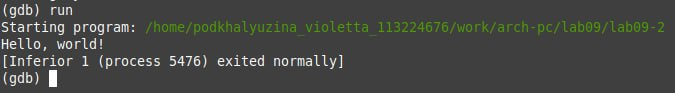
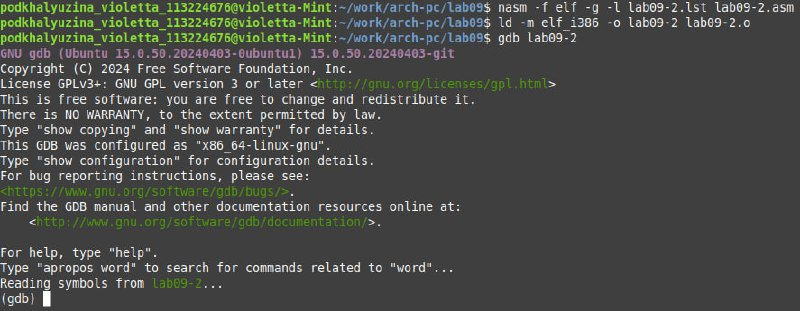
Далее я изменила текст программы, добавив подпрограмму \_subcalcul в подпрограмму \_calcul, для вычисления выражения 𝑓(𝑔(𝑥)), где 𝑥 вводится с клавиатуры, 𝑓(𝑥) = 2𝑥 + 7, 𝑔(𝑥) = 3𝑥 − 1. Т.е. 𝑥 передается в подпрограмму \_calcul из нее в подпрограмму \_subcalcul, где вычисляется выражение 𝑔(𝑥), результат возвращается в \_calcul и вычисляется выражение

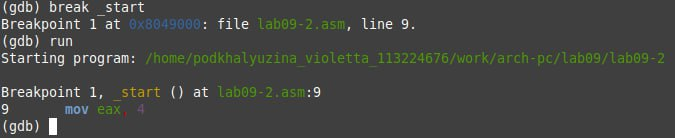


Я создала файл lab09-2.asm с текстом программы из Листинга 9.2



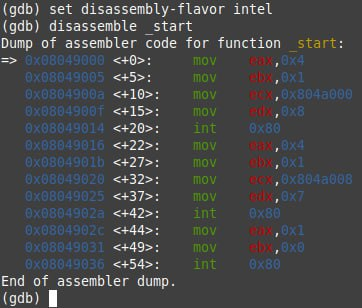
Для работы с GDB в исполняемый файл я добавила отладочную информацию, для этого трансляцию программ необходимо проводить с ключом ‘-g’, а после загрузила исполняемый файл в отладчик gdb.

Далее провесила работу программы, запустив ее в оболочке GDB с помощью команды run (сокращённо r). Для более подробного анализа программы установила брейкпоинт на метку \_start, с которой начинается выполнение любой ассемблерной программы, и запустила её

После я посмотрела дисассимилированный код программы с помощью команды disassemble начиная с метки \_start

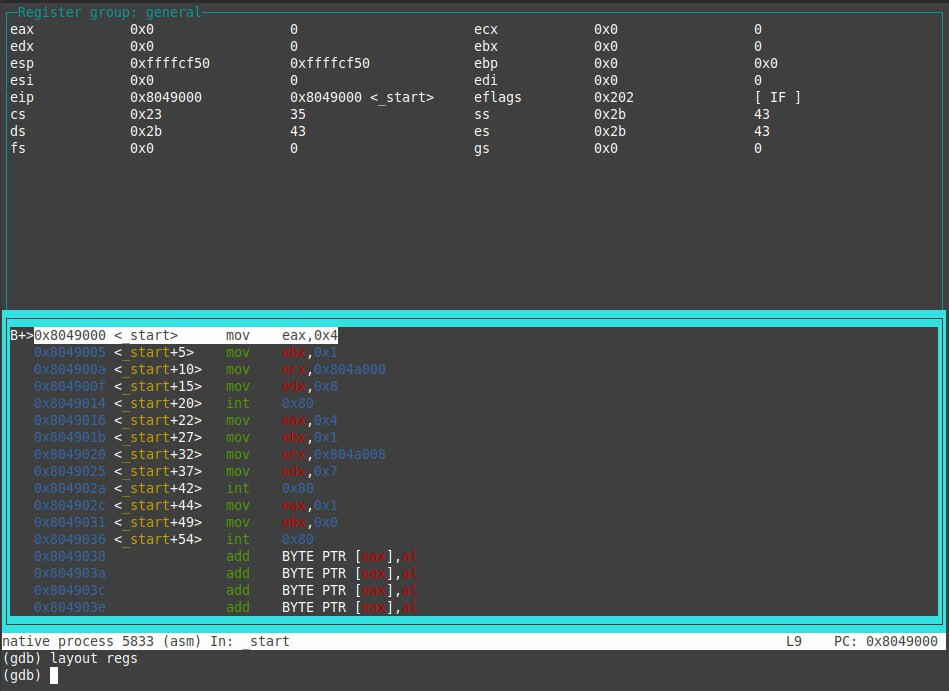
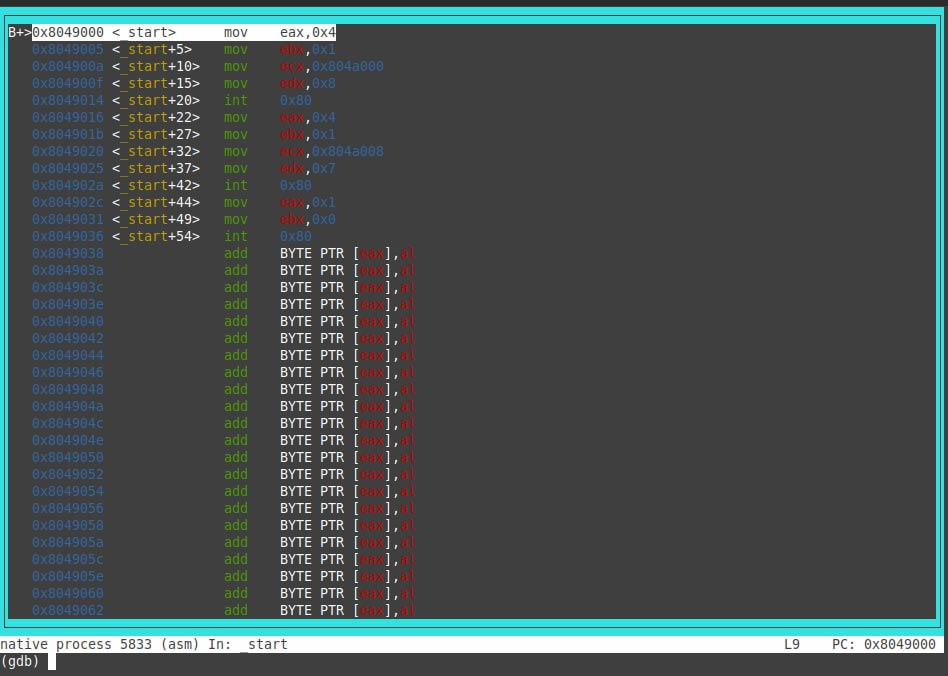


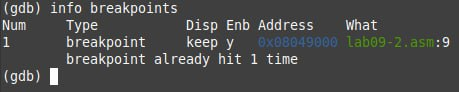
И переключилась на отображение команд с Intel’овским синтаксисом, введя команду set disassembly-flavor intel



Различия между синтаксисом в ATT - pазмер операндов указывается явно с помощью суффиксов, непосредственные операнды предваряются символом $, а у Intel размер операндов неявно определяется контекстом, как ax, eax, непосредственные операнды пишутся напрямую, в АTT имена регистров предваряются символом %, у Intel - имена регистров пишутся без префиксов.

После я включила режим псевдографики для более удобного анализа программы

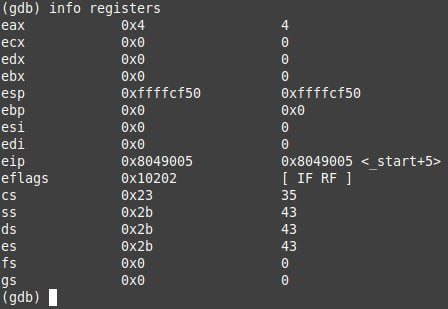
На предыдущих шагах была установлена точка останова по имени метки (\_start). Проверила это с помощью команды info breakpoints



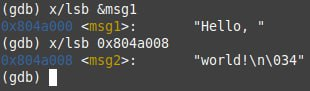
Использую адрес предпоследней инструкции и смотрю информацию о всех установленных точках останова



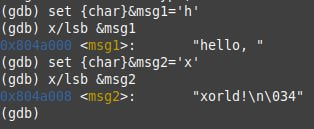
После я посмотрела содержимое регистров также можно с помощью команды info registers (или i r)



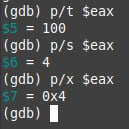
Для отображения содержимого памяти можно использовать команду x <адрес>, которая выдаёт содержимое ячейки памяти по указанному адресу. Формат, в котором выводятся данные, можно задать после имени команды через косую черту: x/NFU <адрес>. С помощью команды x &<имя переменной> также можно посмотреть содержимое переменной. Я посмотрела значение переменной msg1 по имени

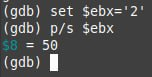


Далее я изменила первый символ переменной msg1



Я вывела в различных форматах значение регистра edx. С помощью команды set изменила значение регистра ebx

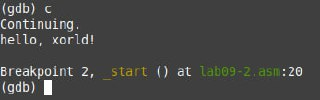


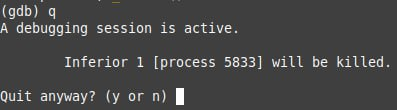




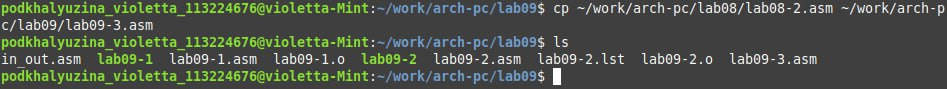
В первом случае выводится порядковый номер символа '2' в таблице ASCII, а во втором случае, так как 2 задана цифрой, а не символом, то и выводится цифра.

Далее я завершила выполнение программы с помощью команды continue (сокращенно c) и вывела из GDB с помощью команды quit (сокращенно q)





После я скопировала файл lab8-2.asm, созданный при выполнении лабораторной работы №8, с программой выводящей на экран аргументы командной строки в файл с именем lab09-3.asm



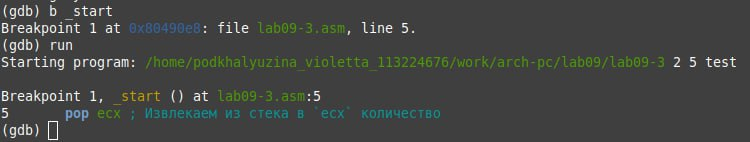
Далее создаю исполняемый файл



Для загрузки в gdb программы с аргументами необходимо использовать ключ --args. А также я загрузила исполняемый файл в отладчик, указав аргументы



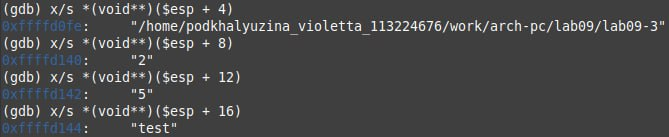
Исследую расположение аргументов командной строки в стеке после запуска программы с помощью gdb. Для начала устанавливаю точку останова перед первой инструкцией в программе и запускаю ее



Адрес вершины стека хранится в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки

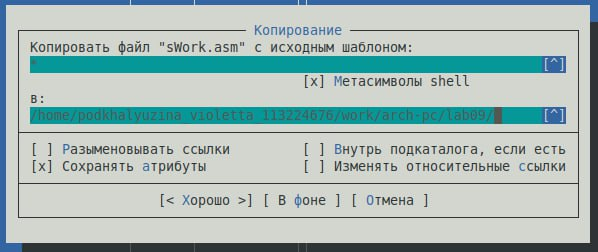


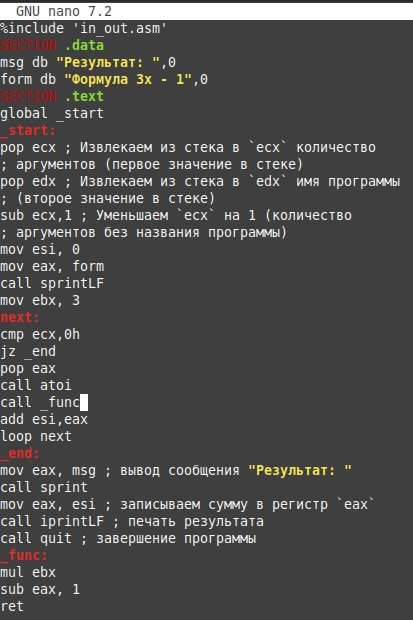
Далее я смотрю остальные позиции стека – по адесу [esp+4] располагается адрес в памяти, где находится имя программы, по адесу [esp+8] храниться адрес первого аргумента, по аресу [esp+12] – второго и так далее

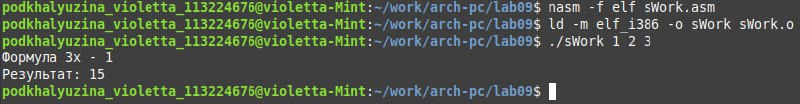
Изменение аргумента команды просмотра регистра esp на +4, число обусловлено разрядностью системы, а указатель void занимает как раз 4 байта, ошибка при аргументе +24 означает, что аргументы на вход программы закончились.

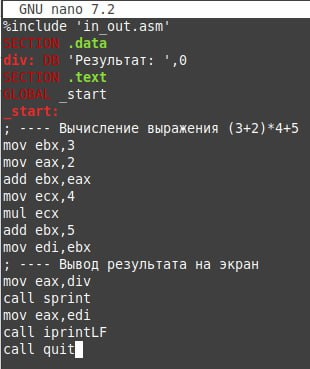
# 3.2 Самостоятельная работа

Сначала я преобразовываю программу из лабораторной номер 8.

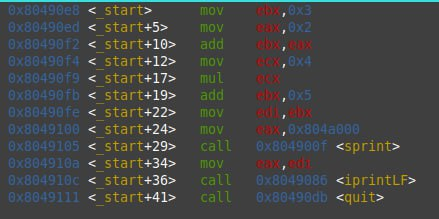


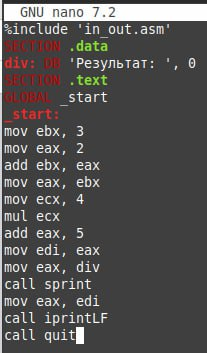


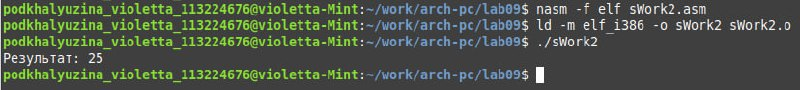
В листинге 9.3 приведена программа вычисления выражения (3 + 2) ∗ 4 + 5. При запуске данная программа дает неверный результат и я это проверила.



После этого с помощью отладчика GDB, анализируя изменения значений регистров, я определила ошибку и исправила ее.

\





Вывод: я преобрела навыки написания программ с использованием подпрограмм, а также познакомилась с методами отладки при помощи GDB и его основными возможностями.

# Контрольные вопросы для самопроверки

1. В ассемблере для оформления и активизации подпрограмм используются инструкции call и ret. Call вызывает подпрограмму, сохраняя адрес возврата в стеке, а ret завершает выполнение подпрограммы, извлекая этот адрес из стека и возвращая управление основной программе.

2. Механизм вызова подпрограмм состоит в том, что инструкция call сохраняет адрес следующей команды (адрес возврата) в стек и передает управление указанной подпрограмме. После завершения выполнения подпрограммы инструкция ret извлекает этот адрес из стека и возвращает управление основной программе, начиная выполнение с команды, следующей за вызовом call.

3. Стек используется для сохранения адреса возврата при вызове подпрограммы. Если подпрограмма сохраняет дополнительные данные в стек, важно восстановить исходное состояние стека перед завершением выполнения, иначе поврежденный адрес возврата приведет к ошибке.

4. Операнд в команде ret определяет количество байт, которые нужно удалить из стека после извлечения адреса возврата. Это используется для очистки параметров вызова, переданных через стек.

5. Отладчик предназначен для поиска и исправления ошибок в программе. Он позволяет выполнять программу по шагам, устанавливать точки останова, отслеживать значения переменных и регистров, а также изменять данные программы во время её выполнения.

6. Отладочная информация связывает исполняемый код с исходным текстом программы, что позволяет анализировать её на уровне строк исходного кода. Для её включения программу нужно компилировать с использованием ключа -g.

7. Breakpoint — это точка останова, при достижении которой выполнение программы приостанавливается. Watchpoint — это точка просмотра, выполнение программы останавливается при изменении или чтении указанной переменной. Checkpoint — это сохраненное состояние программы, которое можно восстановить. Catchpoint — это точка перехвата, срабатывающая при определенных событиях, например, исключениях. Call stack — это стек вызовов, отображающий текущую цепочку активных функций или процедур.

8. Основные команды отладчика gdb включают run (r) для запуска программы, break (b) для установки точки останова, continue (c) для продолжения выполнения после останова, step (s) и next (n) для пошагового выполнения программы, info (i) для получения информации о состоянии программы, print (p) для вывода значений переменных, quit (q) для выхода из отладчика.



# 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/Learningbash-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, 2017.

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. — БХВПетербург, 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, 2015. — 1120 с. — (Классика Computer Science).